Python 3: The Python Environment

Making Sense of User Inputs

Welcome to the O'Reilly School of Technology's (OST) **Python Environment** course! We're happy you've chosen to learn Python programming with us.

Course Objectives

When you complete this course, you will be able to:

- parse command-line arguments and perform string validation.
- build sophisticated structures like bunch classes.
- create your own APIs.
- enhance your code with iterables, iterators, and generators.
- manipulate textual data with regular expressions.
- apply advanced object-oriented programming techniques to Python development.
- exchange binary data with other languages and systems.
- configure user setups and log activity.
- calculate date and time.

By the time you finish the course, you will have expanded your knowledge of Python and applied it to some really interesting technologies.

When you complete this course, you will be able to:

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look like this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

Before you start programming in Python, let's review a couple of the tools you'll be using. If you took **Introduction to Python** and/or **Getting More Out of Python**, you can skip to the next section if you like, or you might want to go through this section to refresh your memory.

About Eclipse

We're using an Integrated Development Environment (IDE) called Eclipse. It's the program filling up your screen right now. IDEs assist programmers by performing tasks that need to be done repetitively. IDEs can also help to edit and debug code, and organize projects.

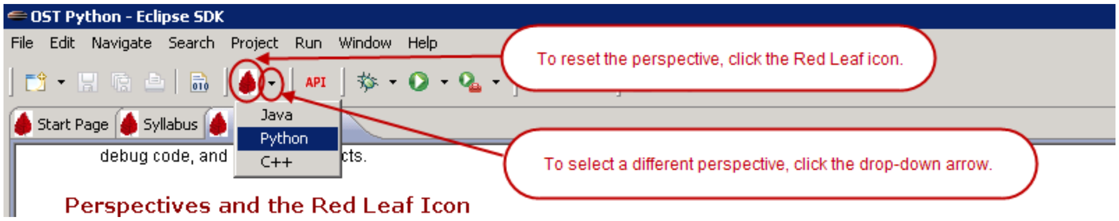
Perspectives and the Red Leaf Icon

The Ellipse Plug-in for Eclipse, developed by the O'Reilly School of Technology, adds a Red Leaf icon to the toolbar in Eclipse. This icon is your "panic button." Because Eclipse is versatile and allows you to move things around, like views, toolbars, and such, it's possible to lose your way. If you do get confused and want to return to the default perspective (window layout), the Red Leaf icon is the fastest and easiest way to do that.

Use the Red Leaf icon to:

- **reset the current perspective:** click the icon.
- **change perspectives:** click the drop-down arrow beside the icon to select different perspectives designed for each course that uses Ellipse.
- **select a perspective:** click the drop-down arrow beside the Red Leaf icon and select the course (**Java**, **Python**, **C++**, etc.). Selecting a specific course opens the perspective designed for that particular course.

  For this course, select **Python**.



To reset the perspective, click the Red Leaf icon.

To select a different perspective, click the drop-down arrow.

Perspectives and the Red Leaf Icon

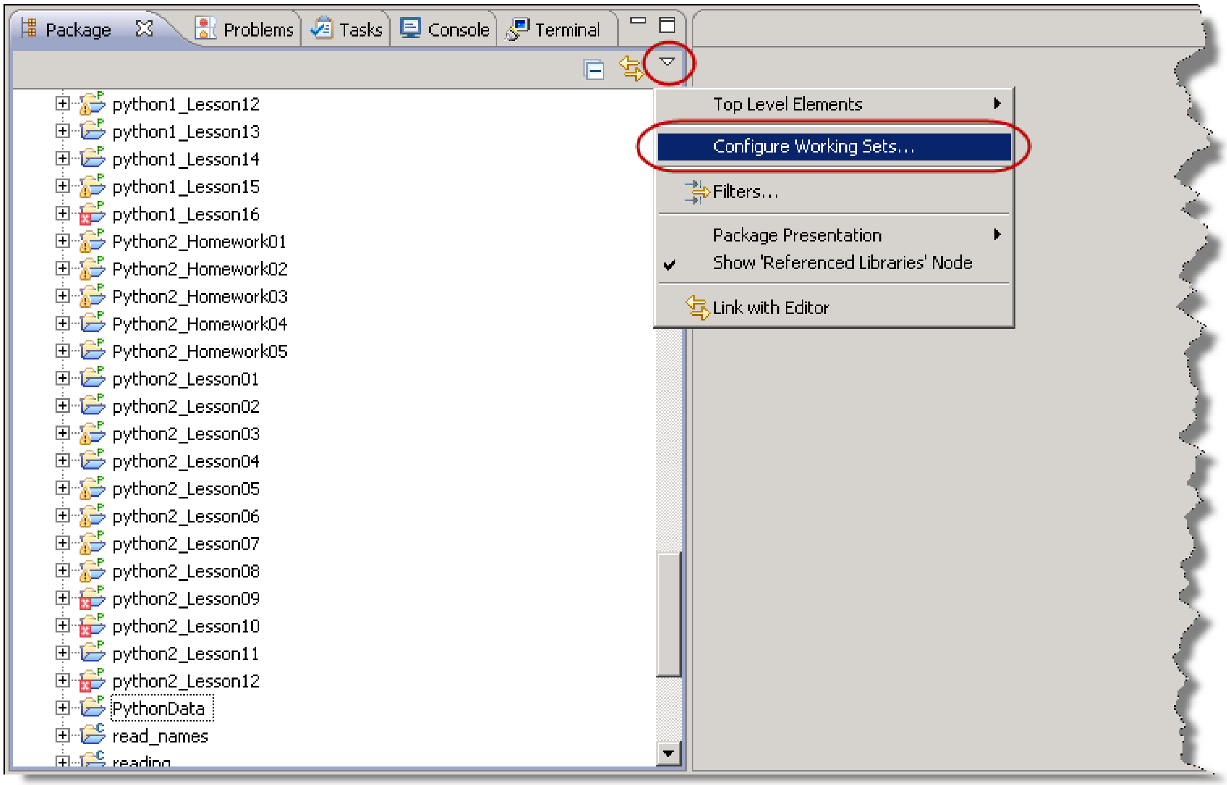
Working Sets

You'll use *working sets* for this course. All projects created in Eclipse exist in the workspace directory of your account on our server. As you create multiple projects for each lesson in each course, your directory could become pretty cluttered. A working set is a view of the workspace that behaves like a folder, but it's actually an association of files. Working sets allow you to limit the detail that you see at any given time. The difference between a working set and a folder is that a working set doesn't actually exist in the file system.

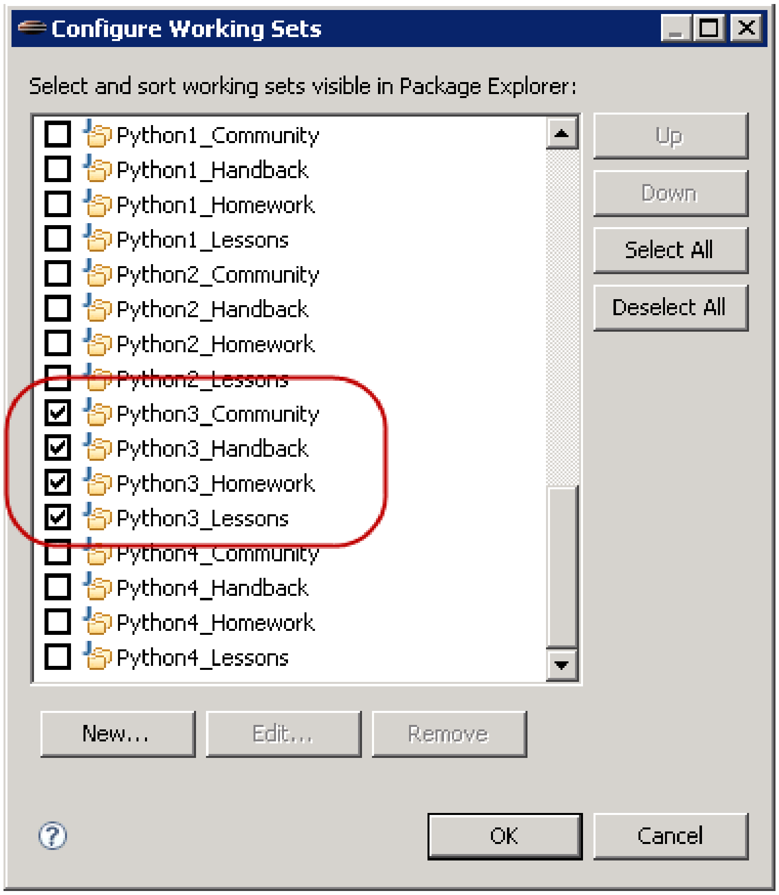
A working set is a convenient way to group related items together. You can assign a project to one or more working sets. In some cases, like the Python extension to Eclipse, new projects are created in a catch-all "Other Projects" working set. To better organize your work, we'll have you assign your projects to an appropriate working set when you create them. To do that, right-click on the project name and select the **Assign Working Sets** menu item.

We've already created some working sets for you in the Eclipse IDE. You can turn the working set display **on** or **off** in Eclipse.

For this course, we'll display just the working sets you need. In the upper right corner of the Package Explorer panel, click the downward-point arrow and select **Configure Working Sets**:



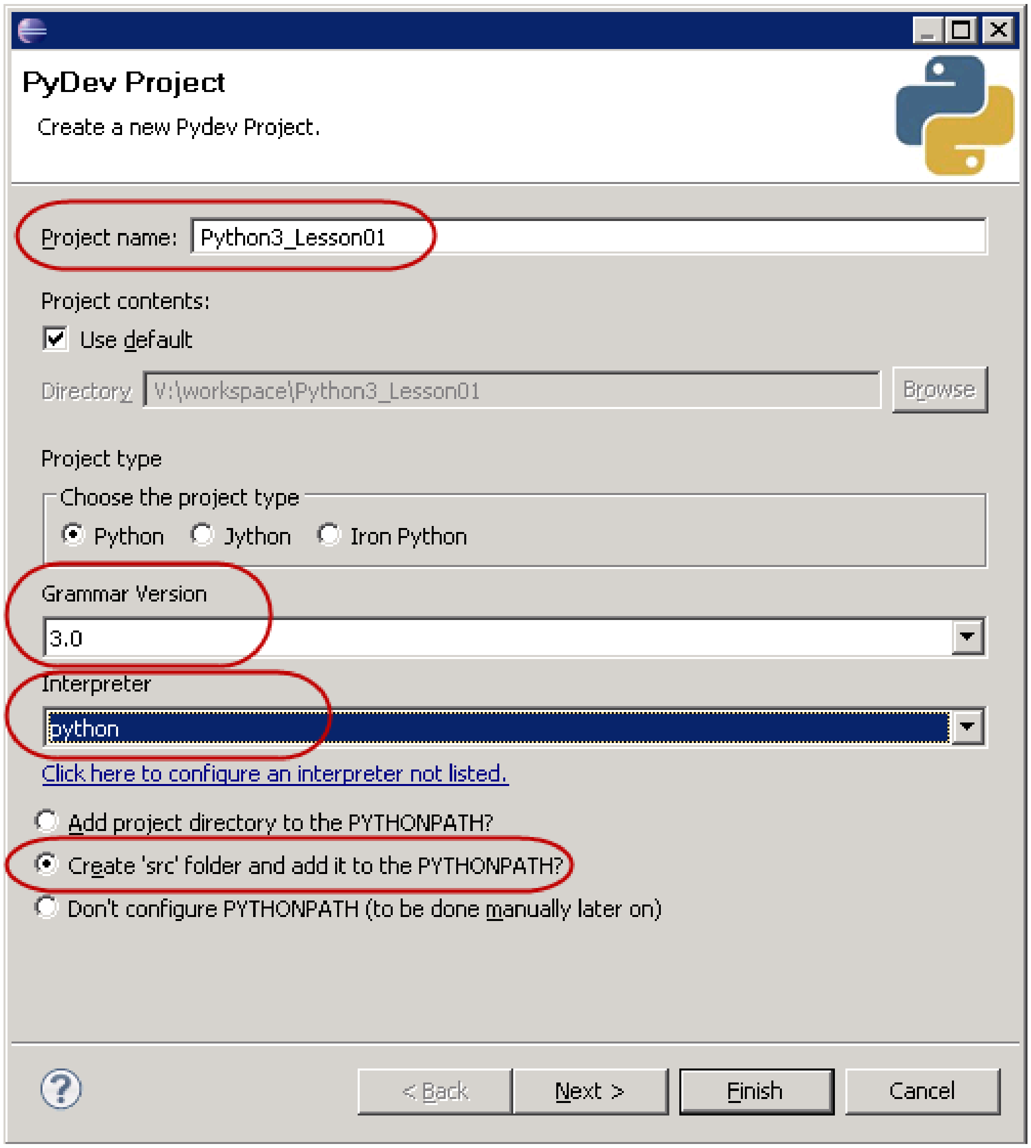
Select the working sets associated with this course (the ones that begin with "Python3"), and click **OK**:



Select and sort working sets visible in Package Explorer:

☐ Python1_Community
☐ Python1_Handback
☐ Python1_Homework
☐ Python1_Lessons
☐ Python2_Community
☐ Python2_Handback
☐ Python2_Homework
☐ Python2_Lessons
☑ Python3_Community
☑ Python3_Handback
☑ Python3_Homework
☑ Python3_Lessons
☐ Python4_Community
☐ Python4_Handback
☐ Python4_Homework
☐ Python4_Lessons

Now, we'll create a project to store our programs for this lesson. Select **File | New | Pydev Project**, and enter information as shown:



PyDev Project

Create a new Pydev Project.

Project name: Python3_Lesson01

Project contents:

☑ Use default

Directory V:\workspace\Python3_Lesson01

Project type

Choose the project type

● Python ○ Jython ○ Iron Python

Grammar Version

3.0

Interpreter

python

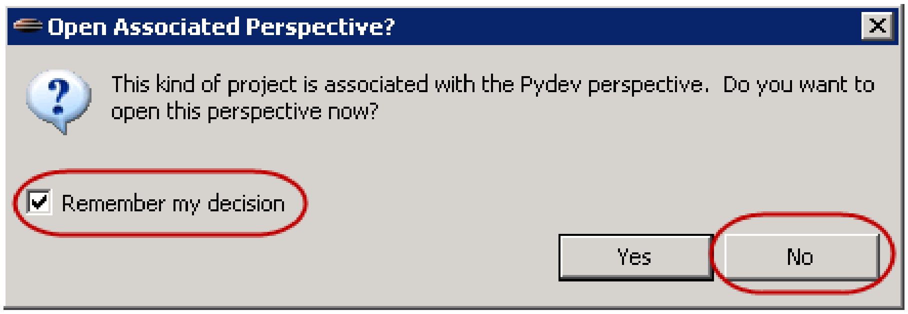
Click here to configure an interpreter not listed.

○ Add project directory to the PYTHONPATH?

● Create 'src' folder and add it to the PYTHONPATH?

○ Don't configure PYTHONPATH (to be done manually later on)

Click **Finish**. When asked if you want to open the associated perspective, check the **Remember my decision** box and click **No**:

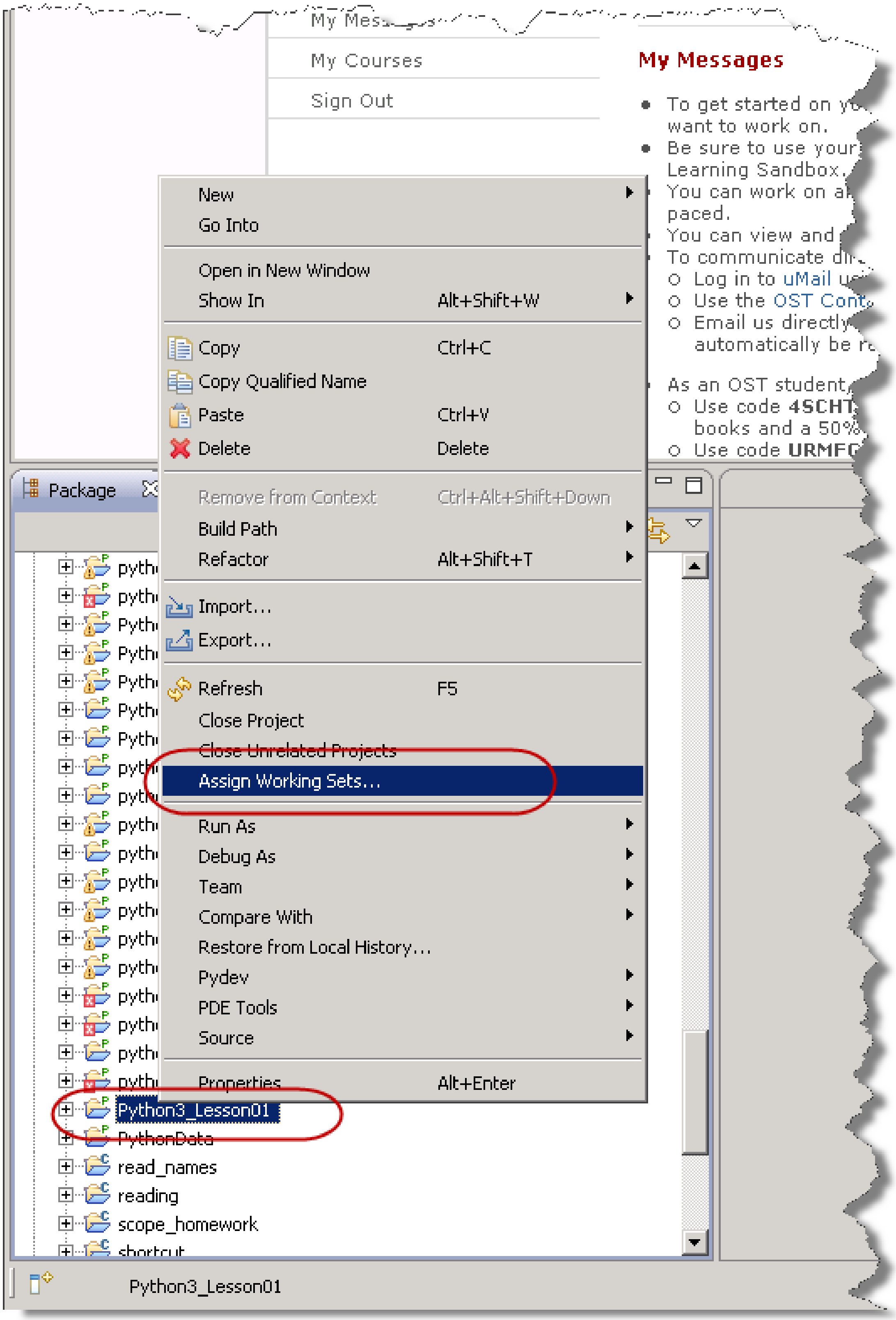


Open Associated Perspective?

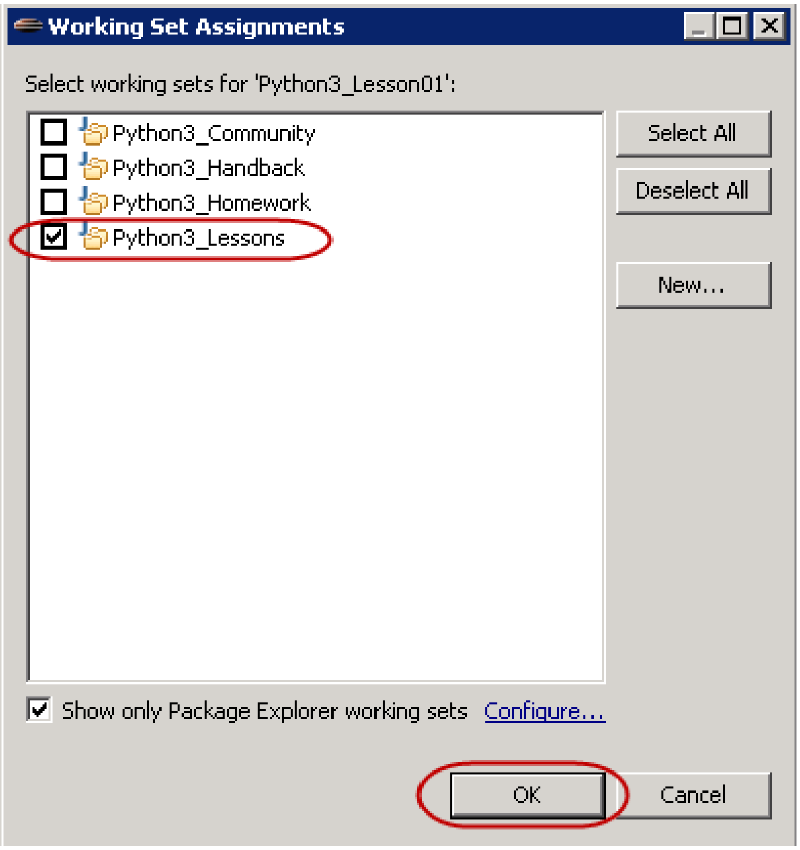
This kind of project is associated with the Pydev perspective. Do you want to open this perspective now?

☑ Remember my decision

By default, the new project is added to the Other Projects working set. Find **Python3_Lesson01** there, right-click it, and select **Assign Working Sets...** as shown:



Select the **Python3_Lessons** working set and click **OK**.



Getting Information for Programs

Programs have to process data. In the preceding two courses, we have used the built-in **input()** function to prompt the user for data we needed. This works well enough for small quantities of data, but would be inconvenient for large amounts. It is much more difficult to write scripts around programs that request data interactively.

Sometimes it's easier for the user, when they are invoking your program by typing a command, to provide information as a part of the command line they enter. Obviously this is most useful for small amounts of data—nobody wants to write an essay at the command line! But for filenames and options (indications to the program of how to modify its processing), the command line is very useful. This also makes writing scripts to use the program much simpler.

Where larger amounts of data are concerned, you frequently get involved in reading textual data and transforming it into other Python types. You have already had to do this when reading numbers via the **input()** function, since that always returns strings. You have to do similar things when reading from files sometimes. The majority of data arrives as text, because much of it is generated by humans.

Data that arrive in textual form need to be transformed into data that the appropriate Python operations can be performed on. So we are going to start this third course in the Python Certificate Series by looking at another way to get data into your programs, and ensure that it can be transformed safely into appropriate Python data types.

Command Line Arguments

The **sys** module contains a number of mechanisms for interacting with the system environment, and **sys.argv** gives you access to the command line the user typed to start the program.

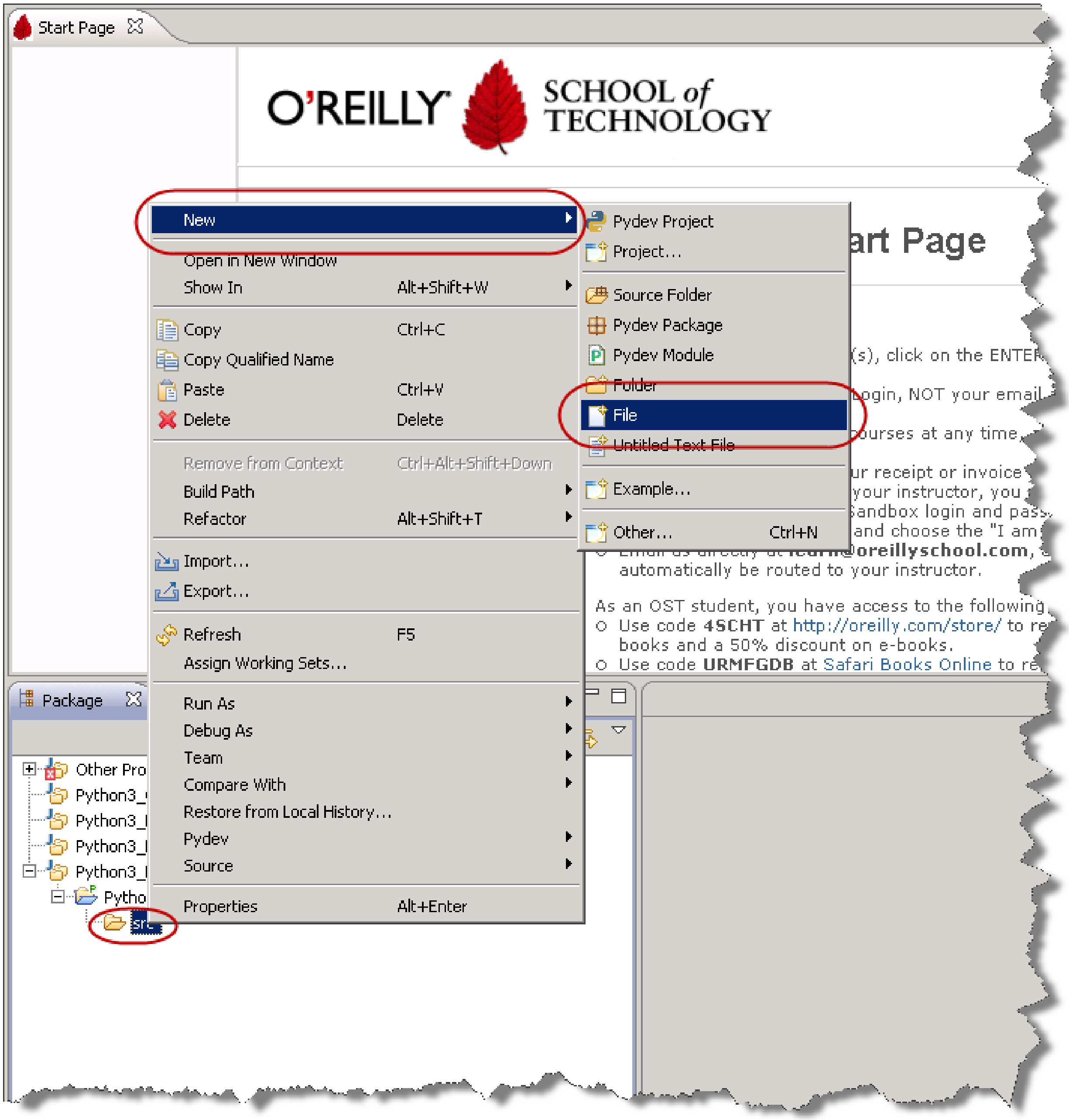
For example, if the user entered the command **python myprog.py one two three**, sys.argv would contain the value **['myprog.py', 'one', 'two', 'three']**. In other words, the program name is **sys.argv[0]**, the first argument to the program call is **sys.argv[1]** and so on.

Emulating the Command Line in Eclipse

Under normal circumstances, you run programs from the command line, but during this class you run them from the Eclipse-based learning system. In previous exercises, there has been no need to examine the command line, and so you started your programs with a simple "Run" command. Now you need to understand how to start a program with a simulated command line.

In order to understand the procedure, we'll create a program that prints out the contents of its command line.

Right-click the **Python3_Lesson01/src** folder in the Package Explorer and select **New | File**:



(s), click on the ENTER

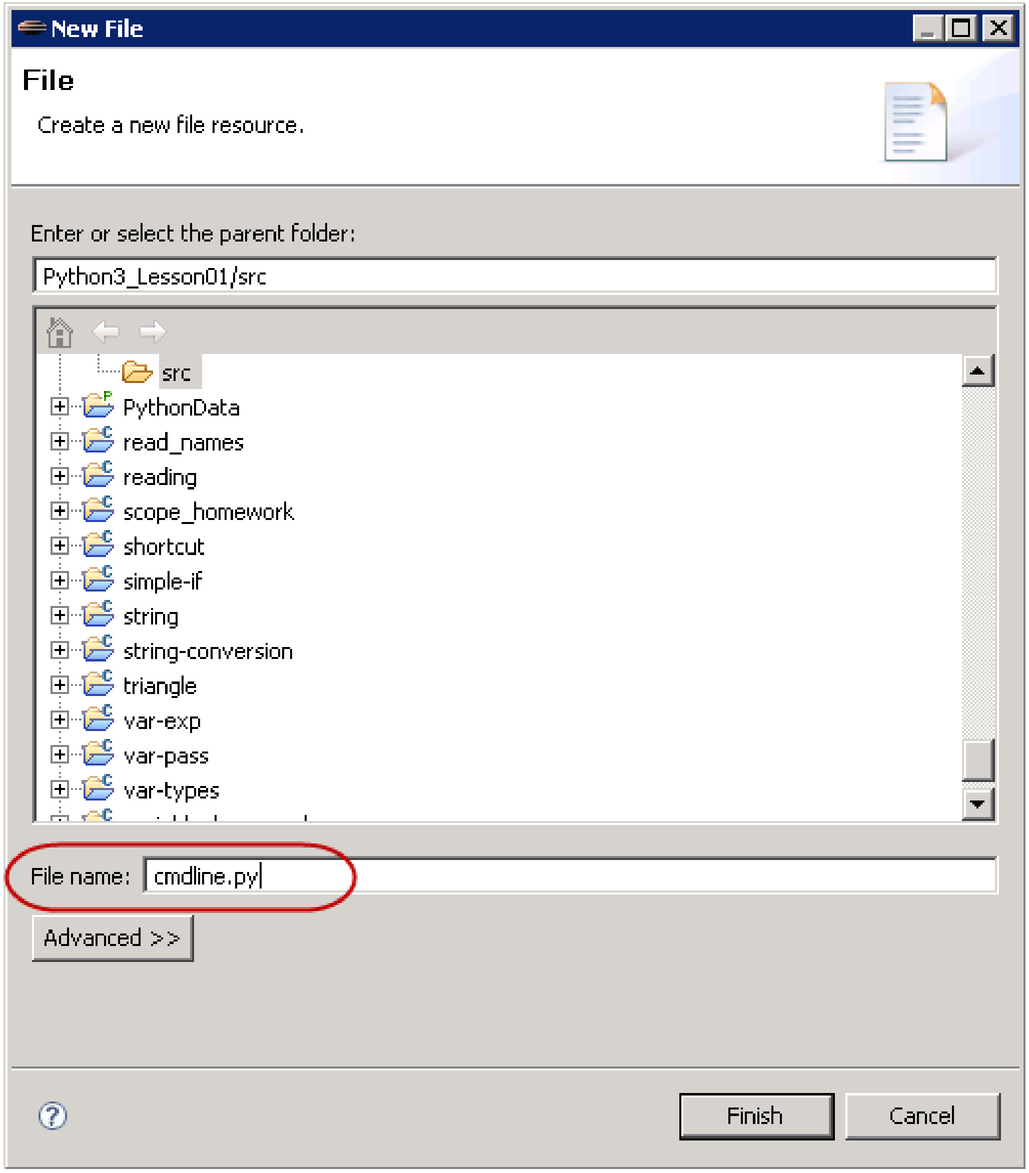
ogin, NOT your email

ourses at any time.

ur receipt or invoice
your instructor, you
Sandbox login and pass
and choose the "I am
oreillyschool.com,
automatically be routed to your instructor.

As an OST student, you have access to the following
- Use code **4SCHT** at http://oreilly.com/store/ to re
books and a 50% discount on e-books.
- Use code **URMFGDB** at Safari Books Online to re

In the New File dialog, enter the name **cmdline.py**, and click **Finish**:



Enter or select the parent folder: Python3_Lesson01/src

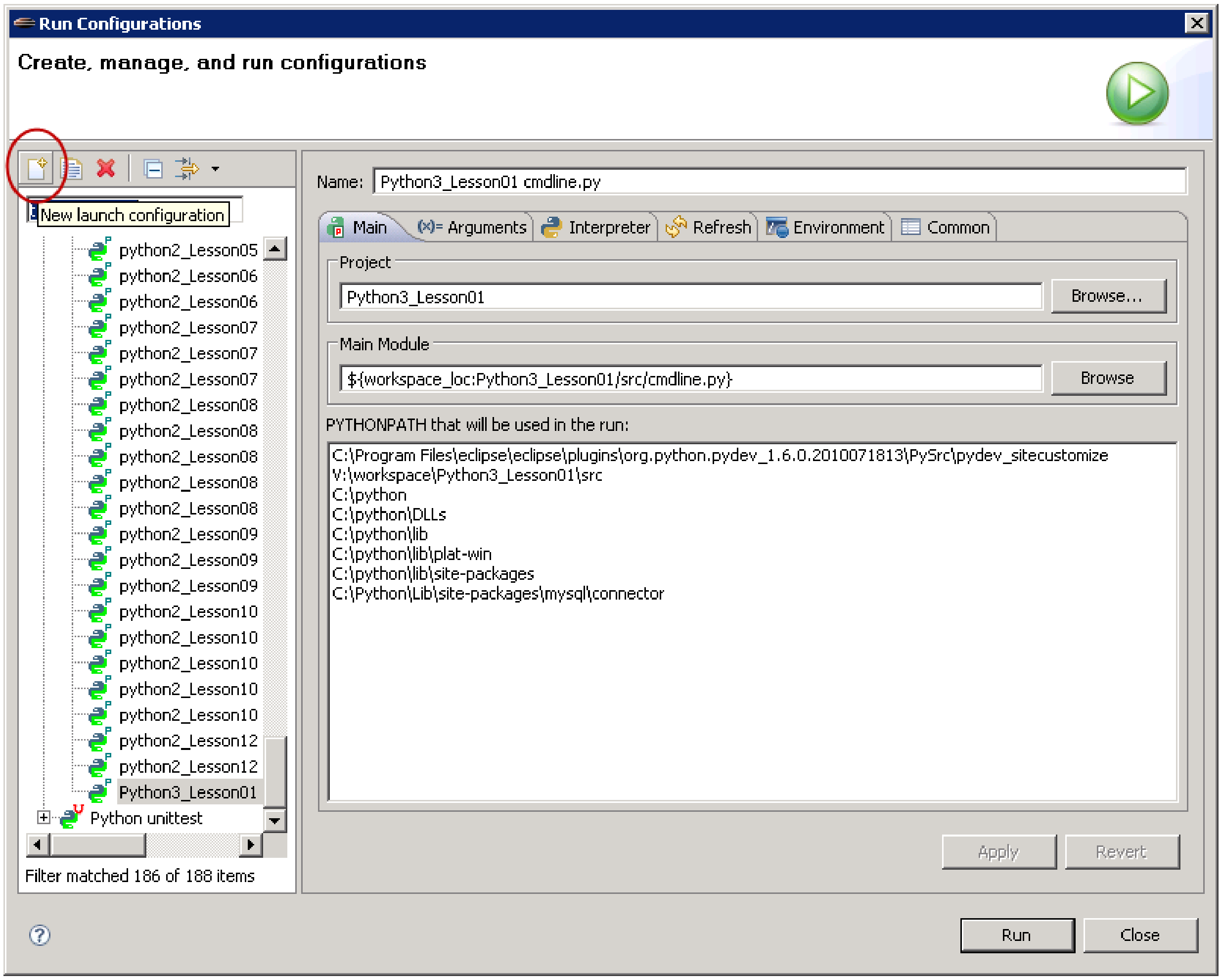
File name: cmdline.py

In the editor, type the code as shown:

CODE TO TYPE: cmdline.py

```
"""
Simple program to dump the command line arguments
"""
import sys
for n,  arg in enumerate(sys.argv):
    print(n, ":", arg)
```

Next we have to tell Eclipse what values to provide for the command line arguments. The easiest way to do this is to create a "Run Configuration", which is a set of specifications of the environment to apply when the program runs. Select **Run | Run Configurations...** from the menu, click the left icon on the Run Configurations dialog toolbar.



Name: Python3_Lesson01 cmdline.py

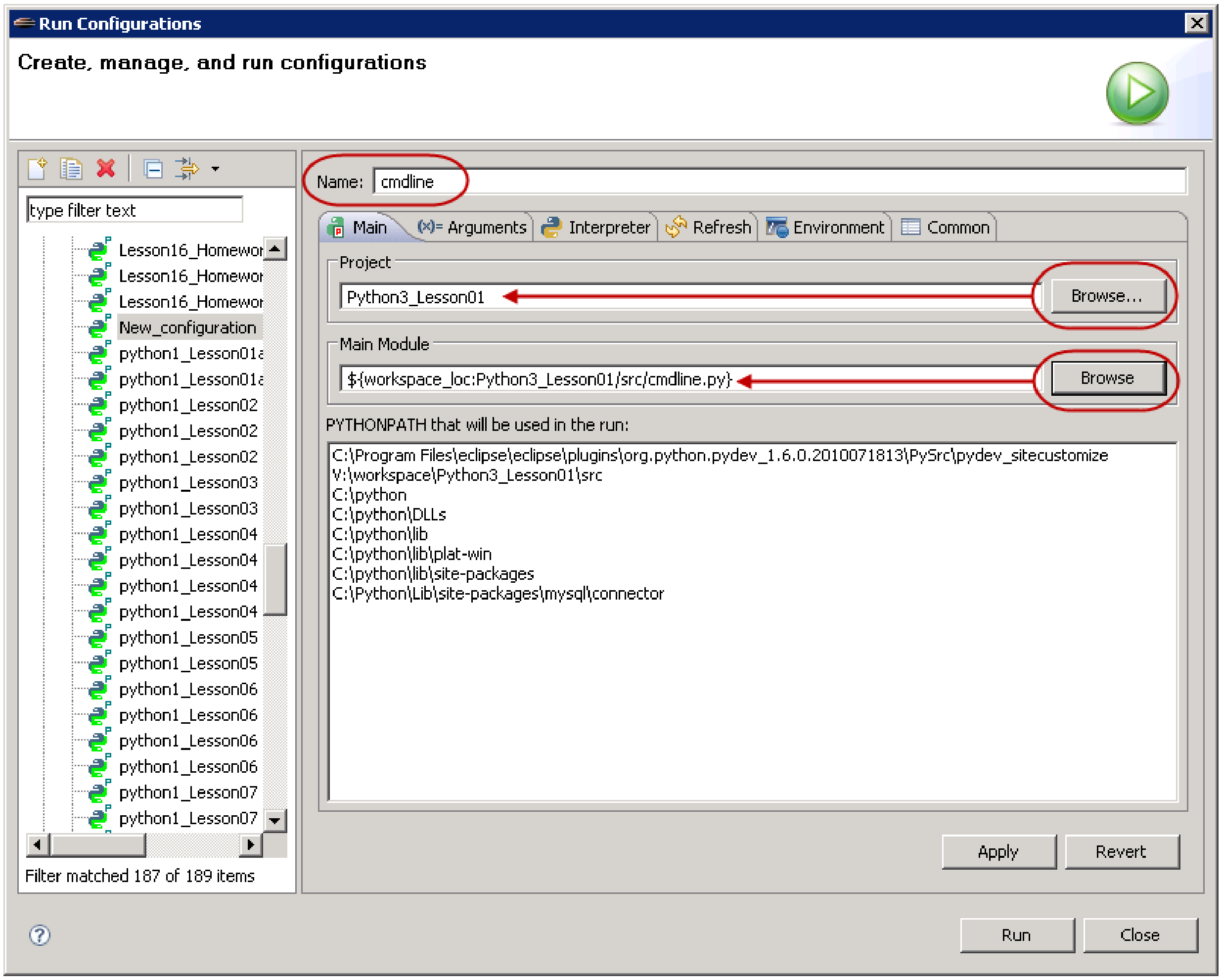
Project: Python3_Lesson01

Main Module: ${workspace_loc:Python3_Lesson01/src/cmdline.py}

PYTHONPATH that will be used in the run:

```
C:\Program Files\eclipse\eclipse\plugins\org.python.pydev_1.6.0.2010071813\PySrc\pydev_sitecustomize
V:\workspace\Python3_Lesson01\src
C:\python
C:\python\DLLs
C:\python\lib
C:\python\lib\plat-win
C:\python\lib\site-packages
C:\Python\Lib\site-packages\mysql\connector
```

This creates a new run configuration, initially named New_configuration, with empty values. Enter **cmdline** in the Name: entry box at the top of the dialog; for the Project, click **Browse** to select the **Python3_Lesson01** project; and for the Main Module, click Browse to select your **cmdline.py** program as the program to run. Observe that Ellipse shows you which directories will be on your Python path.



Name: cmdline

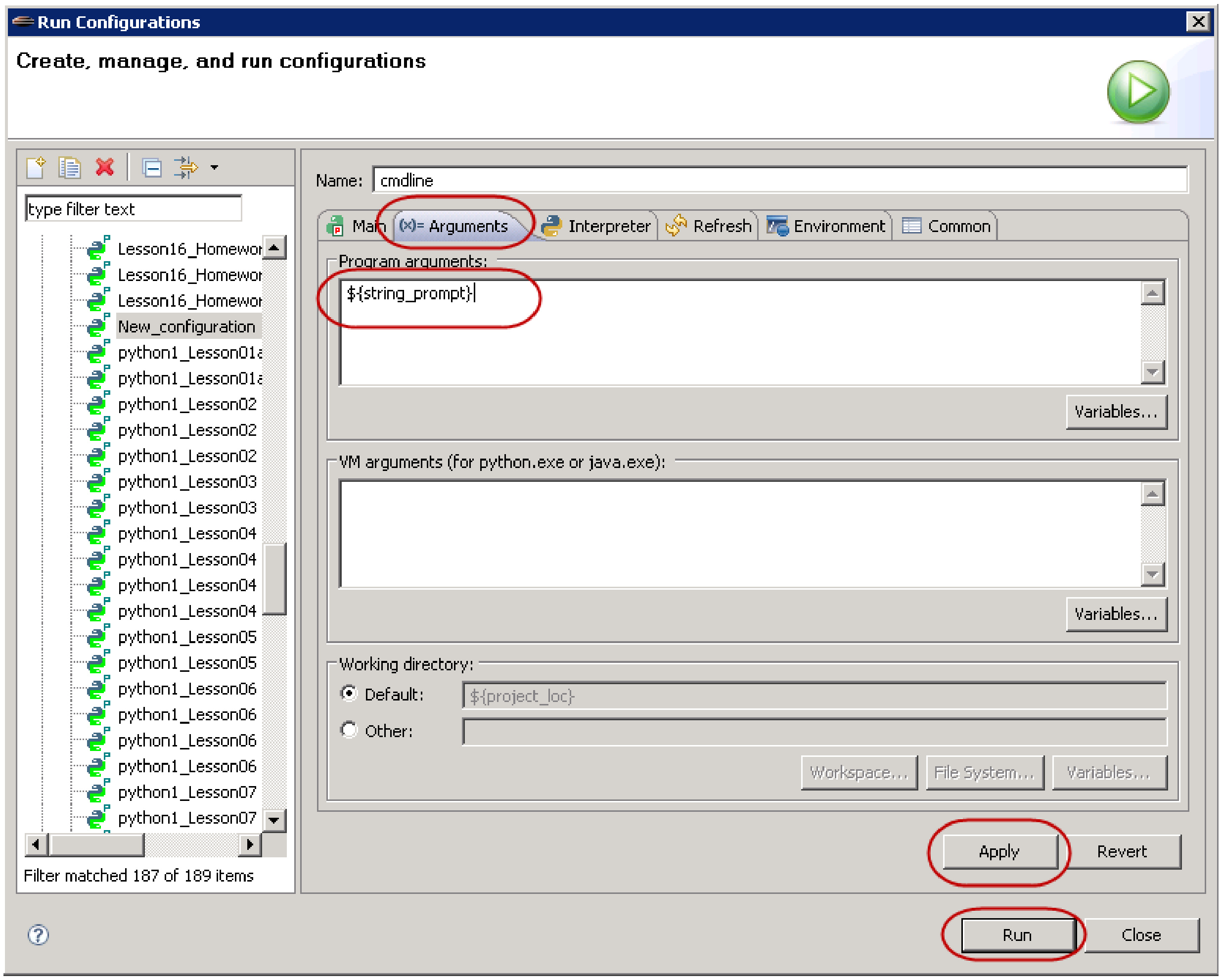
Project: Python3_Lesson01

Main Module: ${workspace_loc:Python3_Lesson01/src/cmdline.py}

PYTHONPATH that will be used in the run:

```
C:\Program Files\eclipse\eclipse\plugins\org.python.pydev_1.6.0.2010071813\PySrc\pydev_sitecustomize
V:\workspace\Python3_Lesson01\src
C:\python
C:\python\DLLs
C:\python\lib
C:\python\lib\plat-win
C:\python\lib\site-packages
C:\Python\Lib\site-packages\mysql\connector
```

Next, select the **Arguments** tab. In the Program Arguments field, enter **${string_prompt}**. This special value tells Ellipse to ask you for the arguments to the program when you run this configuration. Leave everything else as it is:



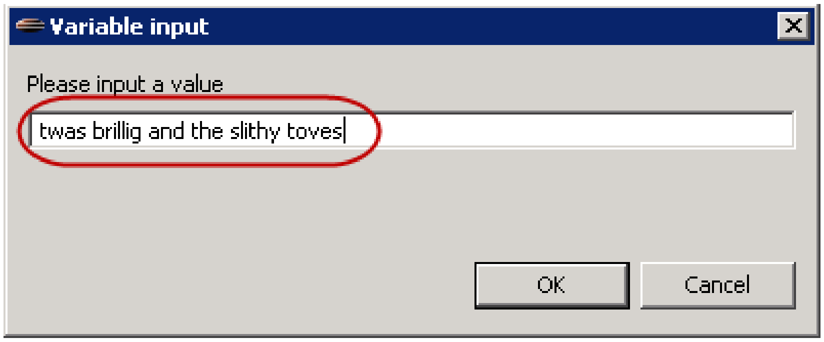
Name: cmdline

Program arguments: ${string_prompt}

VM arguments (for python.exe or java.exe):

Working directory:
● Default: ${project_loc}
○ Other:

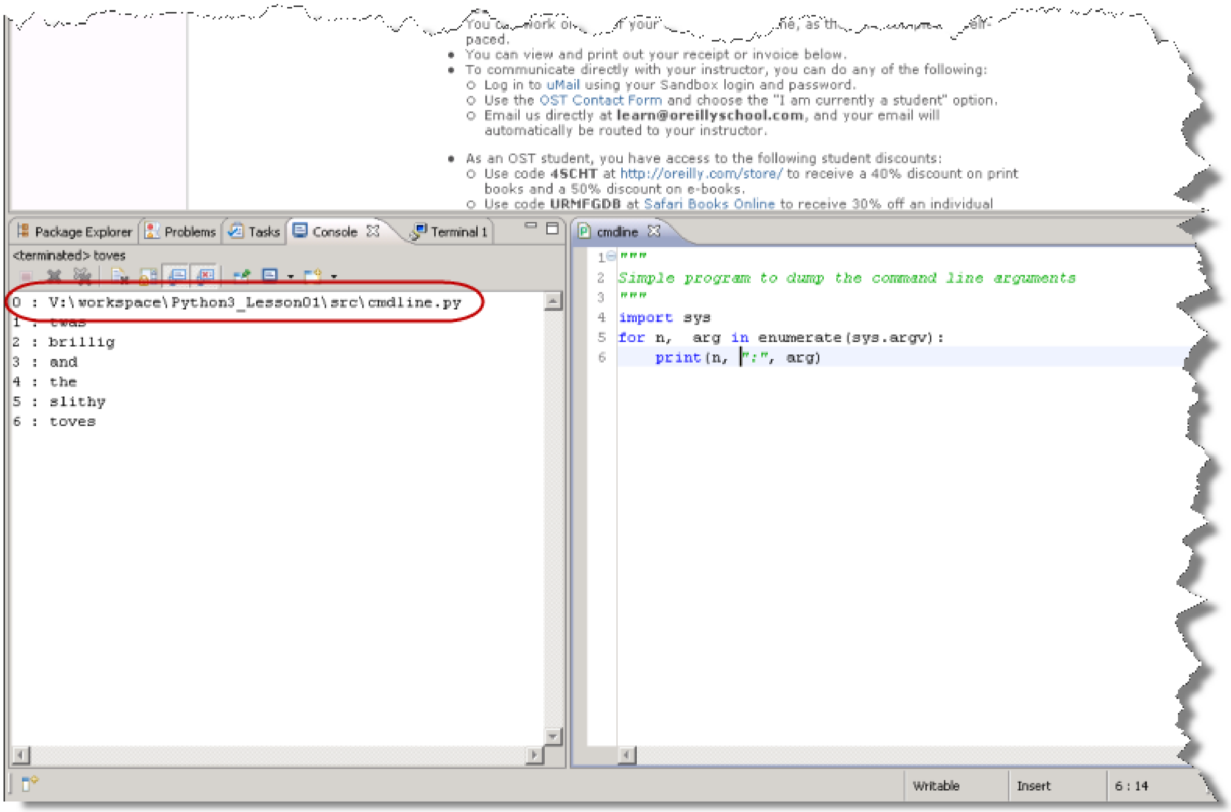
Click **Apply** to save this run configuration, and then click **Run**. You will see a new dialog box, entitled "Variable Input" appear. Enter several words in the dialog box separated by one or more spaces:



Please input a value

twas brillig and the slithy toves

Click **OK**. In the Console tab on the left, you should see the output from this run with the arguments you entered.



- You ca work o f your ne, as th elf-paced.
- You can view and print out your receipt or invoice below.
- To communicate directly with your instructor, you can do any of the following:
  - Log in to uMail using your Sandbox login and password.
  - Use the OST Contact Form and choose the "I am currently a student" option.
  - Email us directly at **learn@oreillyschool.com**, and your email will automatically be routed to your instructor.
- As an OST student, you have access to the following student discounts:
  - Use code **45CHT** at http://oreilly.com/store/ to receive a 40% discount on print books and a 50% discount on e-books.
  - Use code **URMFGDB** at Safari Books Online to receive 30% off an individual

<terminated> toves

```
0 : V:\workspace\Python3_Lesson01\src\cmdline.py
1 : twas
2 : brillig
3 : and
4 : the
5 : slithy
6 : toves
```

```python
"""
Simple program to dump the command line arguments
"""
import sys
for n,  arg in enumerate(sys.argv):
    print(n, ":", arg)
```

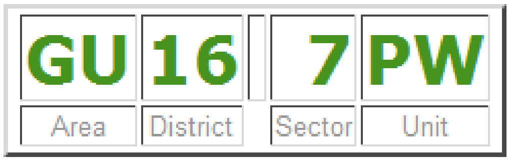
Now you know how to access the command line arguments inside your program. When you run programs outside of Ellipse, you'd just put the data values on the command line (for example, **cmdline.py twas brillig and the slithy toves**) instead of having to run the program and then respond to a prompt.

String Analysis and Manipulation

You have already learned quite a lot about Python strings, and this knowledge will be useful when it comes to accepting data from the user and ensuring, before you try to use it in calculations or for other purposes, that it is appropriate for the intended use.

Data Validation

Ideally, a program should never use data inputs from the user without first checking their reasonableness. Quite often, you need to validate input data by verifying that it conforms to a specific pattern. For example, US ZIP codes are either five digits (the older short form) or nine digits with a dash between the fifth and sixth digit. UK postal codes are somewhat more complicated, with two groups of characters separated by a space. The first group is one or two letters followed by one or two digits, the second group is always one digit followed by two letters:



Other validations might require not only that inputs are numbers, but that they fall within a specific range. The methods of Python's string objects, together with the ability to "carve up" a string using slicing, can be used to perform a limited analysis of a string's contents. If these techniques do not suffice, we need to "bring out the big guns" and use regular expressions, which you will learn about in due course.

For a validation routine, you might decide to return True if the data is acceptable and False if it is not. That approach makes it difficult for the user, though. It is less than helpful to tell them "something is wrong with this data"—you need to explain *what* is wrong with it, and ideally, how they can fix it. This, in turn, means that you have to have some way of getting error indications back from the validation process.

One simple way of validating is to write a function that returns an error message if something is wrong, or None when there are no problems with the data. Once you have saved the result of the function, you can test it (immediately or later) and display the error message if appropriate. You need to be careful with naming of such functions. The result they return will test as True when errors are detected, so use a name like **data_errors()** rather than **verify_data()**, because when the function returns a value it signifies there are errors in the data.

If you want your error checking to be particularly complete, you might want to return more than one error message about a particular piece of data. The natural way to return this would be to accumulate a list inside the validation function and then return the list. If the list is empty, the data is valid. You will see examples of various techniques in the remainder of this lesson.

Testing Strategy

The primary issue with testing validation routines is that the routines are designed to succeed or fail according to the "goodness" of the input data. You therefore need to test both that correct data are correctly validated and that incorrect data are correctly declared invalid.

This means you need two kinds of tests: you have to test that the function fails on bad data, and that it succeeds on good data. If it doesn't do both of these things, it isn't working.

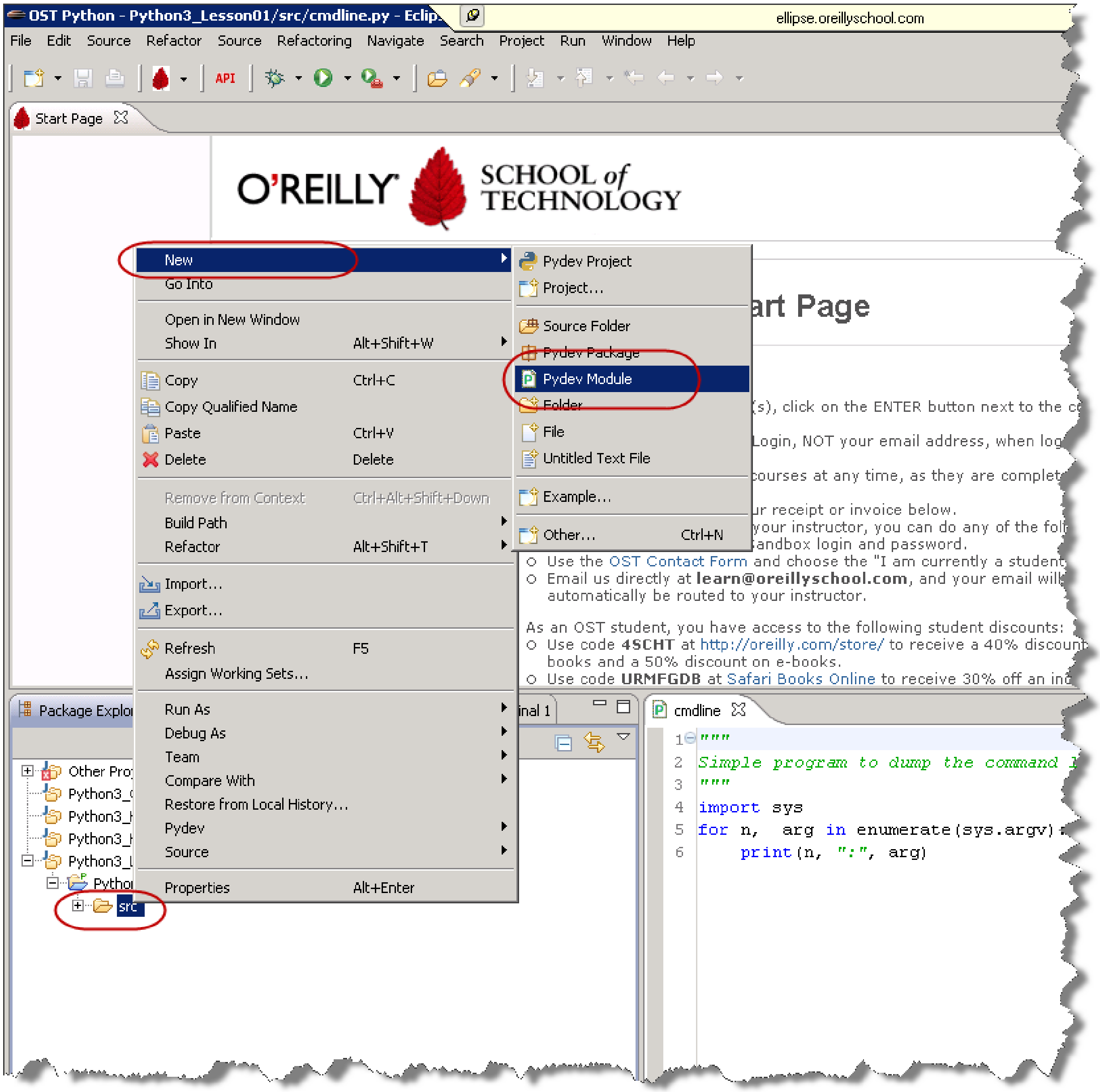
Zip Code Validation

Suppose that you want to verify that a string contains an acceptable US zip code. This kind of task can be puzzling, but it is worth trying to work out for yourself the logic you would apply. The most straightforward and readable way is usually the best—don't worry about efficiency unless you experience a performance problem (you usually won't).

In this particular case, the conditions are fairly easily stated: The zip code must be a string of length five or ten characters. The first five must be numeric; if the length of the string is ten, the sixth character must be a minus sign and the last four must be numeric. Before you get carried away, though, think about how you are going to provide this functionality. Since a zip code check might be useful in all sorts of contexts it probably makes sense to write a function, in a module on its own (you can add other address checking functionality later).

Next, you need to decide on an API for your verification function and write some tests for it. For simplicity, let's just say that it returns a single error message when it finds a problem with the zip code. Remember, if a function continues execution until it "drops off the end," the call will automatically return None, indicating success.

You will start, as usual, by writing the tests. This time we are going to get as much help from Ellipse as possible. In the Package Explorer, right-click the **Python3_Lesson01/src** folder, and select **New | Pydev Module** from the context menu.



(s), click on the ENTER button next to the c
Login, NOT your email address, when log
courses at any time, as they are complet
ur receipt or invoice below.
your instructor, you can do any of the fol
andbox login and password.

- Use the OST Contact Form and choose the "I am currently a student
- Email us directly at **learn@oreillyschool.com**, and your email will automatically be routed to your instructor.

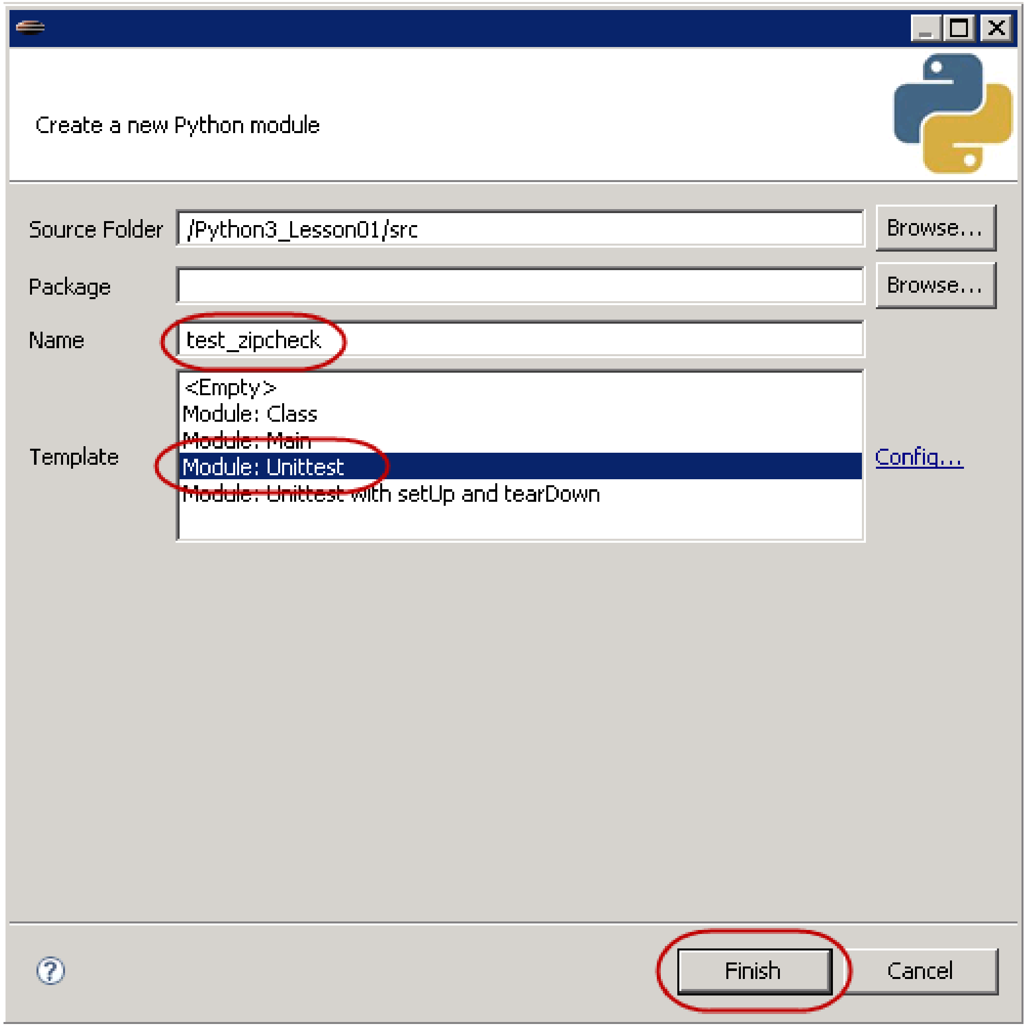
As an OST student, you have access to the following student discounts:

- Use code **4SCHT** at http://oreilly.com/store/ to receive a 40% discount books and a 50% discount on e-books.
- Use code **URMFGDB** at Safari Books Online to receive 30% off an ind

```
"""
Simple program to dump the command l
"""
import sys
for n, arg in enumerate(sys.argv) :
    print(n, ":", arg)
```

In the dialog that appears:

- Leave the Package field blank.
- For Name, enter **test_zipcheck** (you don't need to add the ".py"—Ellipse knows that is needed).
- For Template, select **Module:unittest** from the list.
- Click **Finish**.



Create a new Python module

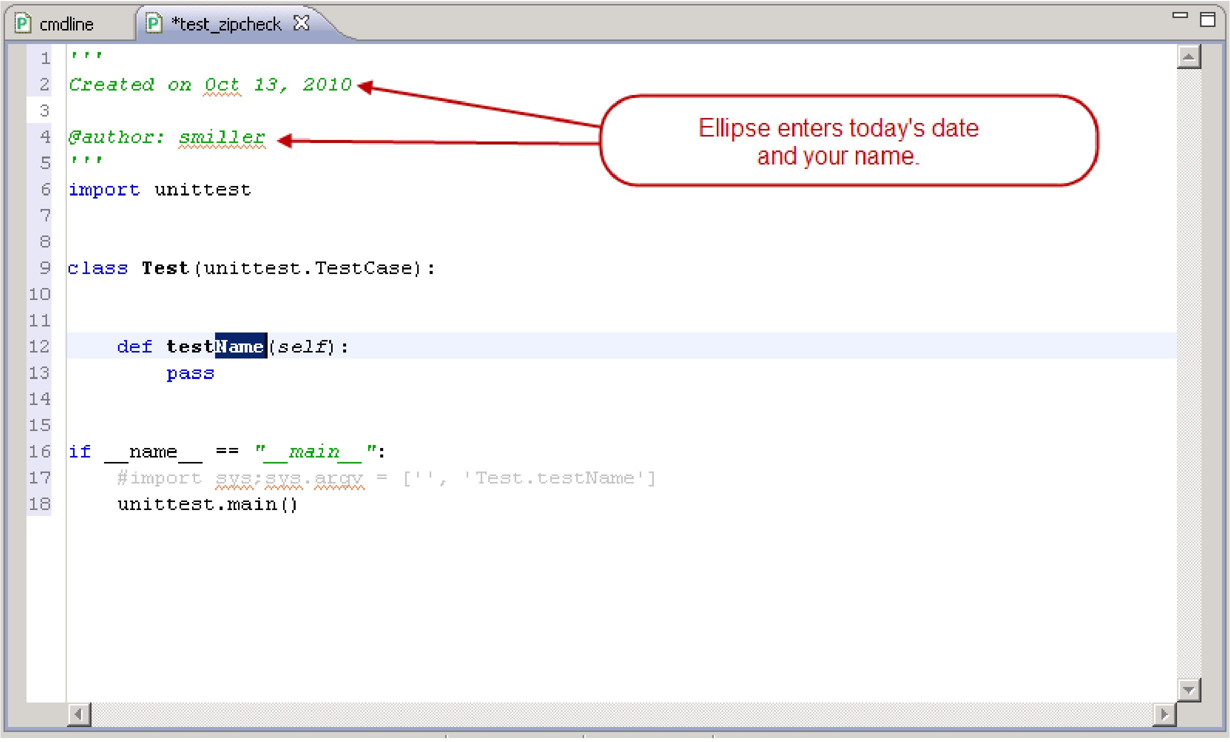
Source Folder: /Python3_Lesson01/src

Package:

Name: test_zipcheck

Template:
- <Empty>
- Module: Class
- Module: Main
- Module: Unittest
- Module: Unittest with setUp and tearDown

This creates a new module with quite a lot of source code already filled in:



```
1  '''
2  Created on Oct 13, 2010
3
4  @author: smiller
5  '''
6  import unittest
7
8
9  class Test(unittest.TestCase):
10
11
12     def testName(self):
13         pass
14
15
16 if __name__ == "__main__":
17     #import sys;sys.argv = ['', 'Test.testName']
18     unittest.main()
```

Ellipse enters today's date and your name.

If you now *immediately* enter **"_zip_errors"** (without the quote marks) you will see that not only is the name of the method completed but the list element in the commented statement (line 17) is also changed. This is a convenience feature from Ellipse—you won't be using the commented statement, but it's neat to see what the software can do.

Next, modify the program by changing the new method's body code and adding another method:

CODE TO EDIT: test_zipcheck.py

```
'''
Created on Aug 29, 2010

@author: sholden

Test the zip_errors() function from the zipcheck module
'''
import unittest
from zipcheck import zip_errors

class Test(unittest.TestCase):

    ~~def testName(self):~~
    def test_zip_errors(self):
        "Tests ensuring errors in data cause validation failures."
        raise TypeError("No tests yet present.")

    def test_zip_successes(self):
        "Test ensuring that valid data passes."
        pass

if __name__ == "__main__":
    #import sys;sys.argv = ['', 'Test.test_zip_errors']
    unittest.main()
```

Before running this program you want to make sure that you at least provide a stub **zip_errors()** function so that your tests fail rather than giving errors when trying to import the function, so create the **zipcheck.py** file as shown below. Note that the stub returns None—although a stub should ideally fail, and the default value of None returned by a stub containing only a **pass** statement will be regarded as successful, you cannot implement a stub that fails when it is supposed to and succeeds when it is supposed to without writing the validation function in all its glory!

CODE TO TYPE: zipcheck.py

```
'''
zipcheck.py: validation function for US zip codes
'''

def zip_errors(z):
    """
    Validate z as either NNNNN or NNNNN-NNNN.
    """
    pass
```

Save and run your **test_zipcheck.py** file now. It shows a failure.

Observe that the test method terminates with the first failure

```
E.
======================================================================
ERROR: test_zip_errors (__main__.Test)
Tests ensuring errors in data cause validation failures.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson01\src\test_zipcheck.py", line 15, in test_zi
p_errors
    raise TypeError("No tests yet present.")
TypeError: No tests yet present.

----------------------------------------------------------------------
Ran 2 tests in 0.016s

FAILED (errors=1)
```

As usual, this is hardly surprising with an empty stub replacing the desired functionality. Note, however, that the second test passes in its entirety. This is because the function has to either succeed or fail, and since by default it succeeds, by default good zip codes are accepted as good.

In fact the second test is there to verify that there are no failures to accept good data. Unless you induce such failures, you will probably never see a failure of this test. If you do, however, you know something serious has gone wrong. Furthermore the first test, being a stub, would have also succeeded if you hadn't specifically made it fail with the **raise** statement.

You can remove the **raise** as soon as you introduce real tests, which is the next step. You are going to add negative tests, which will fail if the validation function affirms data acceptable when it should not be, and positive tests, which will fail if the function refuses to accept a string when it should.

This is a matter of balance. For now, leave your stub function as it is and make the tests a little more comprehensive.

CODE TO EDIT: test_zipcheck.py

```
'''
Created on Aug 29, 2010

@author: sholden

Test the zip_errors() function from the zipcheck module
'''
import unittest
from zipcheck import zip_errors

class Test(unittest.TestCase):

    def test_zip_errors(self):
        "Tests ensuring that errors in data cause validation failures."
        ~~raise TypeError("No tests yet present.")~~
        self.assertIsNotNone(zip_errors("1234"), "Accepting length 4")
        self.assertIsNotNone(zip_errors("12345-678"), "Accepting length 9")
        self.assertIsNotNone(zip_errors("1234e"), "Accepting alphabetic 5")
        self.assertIsNotNone(zip_errors("12345-678Y"), "Accepting alphabetic 5+4
")
        self.assertIsNotNone(zip_errors("12345/6789"), "Accepting non-hyphen")

    def test_zip_successes(self):
        "Test ensuring that valid data passes."
        ~~pass~~
        self.assertIsNone(zip_errors("12345"), "Not accepting 5-digit zips")
        self.assertIsNone(zip_errors("12345-6789"), "Not accepting 9-digit zips"
)

if __name__ == "__main__":
    #import sys;sys.argv = ['', 'Test.test_zip_errors']
    unittest.main()
```

Save and run the test. We still see a failure, but now at least we can see that zip codes of incorrect length are being caught.

OBSERVE: With real tests in there, the first test still fails

```
F.
======================================================================
FAIL: test_zip_errors (__main__.Test)
Tests ensuring that errors in data cause validation failures.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson01\src\test_zipcheck.py", line 15, in test_zi
p_errors
    self.assertIsNotNone(zip_errors("1234"), "Accepting length 4")
AssertionError: Accepting length 4

----------------------------------------------------------------------
Ran 2 tests in 0.015s

FAILED (failures=1)
```

So now we need to enhance **zipcheck** to test the length of the input. There are only two valid values.

CODE TO EDIT: zipcheck.py

```
'''
zipcheck.py: validation function for US zip codes
'''

def zip_errors(z):
    """
    Validate z as either NNNNN or NNNNN-NNNN.
    """
    ~~pass~~
    l = len(z)
    if l not in (5, 10):
        return "Zip codes should be 5 or 10 characters long"
    return
```

Save and run the test. Notice that the validation function now accepts an input as valid if it doesn't specifically find anything wrong with it. This requires your error checks to be exhaustive (which they aren't at the moment, as you discover by running your tests again).

OBSERVE: Running test_zipcheck.py shows length checks are working

```
F.
======================================================================
FAIL: test_zip_errors (__main__.Test)
Tests ensuring that errors in data cause validation failures.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson01\src\test_zipcheck.py", line 17, in test_zi
p_errors
    self.assertIsNotNone(zip_errors("1234e"), "Accepting alphabetic 5")
AssertionError: Accepting alphabetic 5

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

You also need to make sure that the first five characters of the zip are all numeric (and for ten-digit inputs, that the last four characters are numeric too). This is a relatively simple modification: you just return an error message complaining about the characters unless they are all numeric. The only slightly tricky part is not testing the last four unless the length of the input is ten.

CODE TO EDIT: zipcheck.py

```
'''
zipcheck.py: validation function for US zip codes
'''

def zip_errors(z):
    """
    Validate z as either NNNNN or NNNNN-NNNN.
    """
    l = len(z)
    if l not in (5, 10):
        return "Zip codes should be 5 or 10 characters long"
    if (not z[:5].isdigit() or
        len(z) == 10 and not z[6:].isdigit()):
        return "Zip code contains non-numeric characters"
    return
```

Save and run the test. Now the function correctly raises errors for zips with non-numeric characters in them, but you still see failures because there is nothing yet that checks to make sure that, in a zip+4, the two parts of the zip are separated by a dash.

OBSERVE: the tests still fail, even though further checks have been added

```
F.
======================================================================
FAIL: test_zip_errors (__main__.Test)
Tests ensuring that errors in data cause validation failures.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson01\src\test_zipcheck.py", line 19, in test_zi
p_errors
    self.assertIsNotNone(zip_errors("12345/6789"), "Accepting non-hyphen")
AssertionError: Accepting non-hyphen

----------------------------------------------------------------------
Ran 2 tests in 0.016s

FAILED (failures=1)
```

The final test makes sure that ten-digit zips have a dash in the correct position. This is the last check that we can make—any zip that passes all those tests is good. If none of the tests detect a failure it's OK to succeed by returning **None**, which as usual happens by default.

CODE TO EDIT: zipcheck.py

```
'''
zipcheck.py: validation function for US zip codes
'''

def zip_errors(z):
    """
    Validate z as either NNNNN or NNNNN-NNNN.
    """
    l = len(z)
    if l not in (5, 10):
        return "Zip codes should be 5 or 10 characters long"
    if (not z[:5].isdigit() or
        (len(z) == 10 and not z[6:].isdigit())):
        return "Zip code has non-numeric characters"
    if l == 10 and z[5] != "-":
        return "Ten-digit zips must have a dash between the two parts"
    return
```

Save and run the test. Finally, it passes! The bad zip codes are returning error messages and the good zip codes aren't.

OBSERVE: Finally the test passes

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Notice that separation between the tests of good zips and the tests of bad zips made it somewhat easier to observe that the test coverage was improving. The fact that the second test always succeeded simply shows that the code was developing along the right lines. Had it failed at any time, you would have seen that the validator was failing to approve valid data, which would have been valuable feedback.

So, that gives you a brief introduction to data validation in Python. Ideally you should *never* use data that has not been through some validation process. Failure to validate inputs is the source of many well-known security issues, including "buffer overflow" attacks and "SQL Injection" attacks. Get in the habit of validating your data, and make sure that you use tested validation routines so you can have a reasonable degree of confidence that they are going to validate as expected.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Converting Data into Structured Objects

In the previous courses, we've touched on various methods of structuring data. Iterators such as **lists** and **tuples** have their place, as does the **dict**. The clever usage of these fundamental structural elements in Python is a defining hallmark of a skilled developer.

Sometimes, though, you need objects that behave in different ways. Assigning one behavior to the data such as '*render to CSV*' can be easy. But what if you have a dozen behaviors to consider? What if you need to include behaviors such as '*print prettily to the screen, sum up the integer values, add an ISBN from the O'Reilly bookstore*', and a dozen more operations?

You could use your **list**, **tuple**, or **dict** structures in combination with a dozen functions to create the functionality you need. However, that isn't very portable, as remembering to import all your functions across multiple modules is error-prone and time-consuming. Really what you are looking for is a way to carry the functions around with the data, making them really easy to apply.

This means it's time for us to revisit object oriented programming. With a little bit of work, you can apply a sound structure to incoming data. Applying this sound structure to your data can provide a number of positive benefits. Since the data is in a predictable format, you can more easily write code to support the data. The structure can also be assigned behaviors, which can be applied to the data. All the behaviors are defined in a class definition, making them readily available to all instances of the class.

If you document the expected structure of the data you expect to receive, you are providing an interface for yourself to follow in the future. The wonderful thing is that Python gives you the tools to easily create interfaces that are easy to understand, flexible, and very powerful.

This lesson includes the following sections:

Constructing classes

In previous courses and lessons, we learned how to write classes and create objects. We also learned about the **__init__()** special constructor method used each time an object is instantiated. Now, we'll learn a few more things about the **__init__()** method and things you can do to better handle behavior of data.

You may remember that instances of your classes normally keep their instance attribute values in a dict known as **self.__dict__**. Remind yourself with a quick interactive interpreter session.

CODE TO TYPE: Enter the following code at an interactive console session

```
>>> class Meter:
...     def __init__(self, voltage):
...         self.limit = voltage
...
>>> m1 = Meter(20)
>>> m1.label = "Apartment 2214"
>>> m1.__dict__
{'limit': 20, 'label': 'Apartment 2214'}
```

See how **self.__dict__** implements the instance m1's local namespace? When we bind a value to the name "limit" in the instance's namespace with self.limit = voltage, a new key "limit" appears in the instance's __dict__, associated with the value "20." One of the reasons why namespaces seem so like dicts is that a dict is often used to implement a namespace.

Introducing the Bunch Class

Through the use of keyword arguments, Python gives us the ability to create a *bunch class*. A bunch class takes incoming data and saves it as attributes. That sounds more sophisticated than it is, so let's write a bunch class and see what it means. Create a new **Python3_Lesson02** project and assign it to your **Python3_Lessons** working set. Then create **bunchclass.py** in your **Python3_Lesson02/src** folder as shown:

CODE TO TYPE: bunchclass.py

```
"""
Simple bunch class
"""
class Bunch(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

if __name__ == "__main__":
    b = Bunch(name="Python 3", language="Python 3.0.1")
    print(b.name)
    print(b.language)
    print(b.__dict__)
```

Save and run it.

OBSERVE: The attribute values and the updated __dict__

```
Python 3
Python 3.0.1
{'name': 'Python 3', 'language': 'Python 3.0.1'}
```

You see the two values, "Python 3" and "Python 3.0.1," printed from the "name" and "language" attributes of the "b" object. But the **Bunch** class lacks those attributes!

Remember that instances keep their attributes in a dict-like object (named **__dict__**), and that the code guarded by **if __name__ == "__main__":** will only be executed if the module is run as a main program, and not when it is imported by some other program. In the latter case, you don't want print statements running in the middle of someone else's program!

Let's take a closer look:

OBSERVE: bunchclass.py

```
"""
Simple bunch class
"""
class Bunch(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

if __name__ == "__main__":
    b = Bunch(name="Python 3", language="Python 3.0.1")
    print(b.name)
    print(b.language)
    print(b.__dict__)
```

This **Bunch** class uses the magic **__dict__** attribute's **update()** method to dynamically add attributes to the object based according to the **keyword arguments** passed into the class. Note that the **__init__()** method's second argument is prefixed by "**", so keyword arguments are collected in a dict named **kwargs**. Calling **__dict__**'s **update()** method copies the keys and values from **kwargs** to **__dict__**.

Note In Python 3, when we define a class, we don't need to specify that it inherits from **object**, but it doesn't hurt to do so. We do it here just to remind you that the **Bunch** class is a child of the built-in **object**.

All our code should have tests, even programs as seemingly simple as this. Convert the above program to use the **unittest** framework.

CODE TO EDIT: bunchclass.py

```
"""
Simple bunch class
"""
import unittest

class Bunch(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

class TestBunch(unittest.TestCase):
    def test_attributes(self):
        b = Bunch(name="Python 3", language="Python 3.0.1")
        self.assertEqual("Python 3", b.name)
        self.assertEqual("Python 3.0.1", b.language)

if __name__ == "__main__":
    ~~b = Bunch(name="Python 3", language="Python 3.0.1")~~
~~    print(b.name)~~
~~    print(b.language)~~
~~    print(b.__dict__)~~
    unittest.main()
```

Save and run it.

OBSERVE: The test passes when you run the program

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The tests are small here, so it is OK to add them to the basic module rather than making a separate test module. The current code always imports the **unittest** module even when it is not going to be used (when the module is imported rather than running as a main program). You can correct this by moving both the import of **unittest** and the code for the test class itself so that these pieces of code are only executed when required:

CODE TO EDIT: bunchclass.py

```
"""
Simple bunch class
"""
~~import unittest~~
class Bunch(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

~~class TestBunch(unittest.TestCase):~~
~~    def test_attributes(self):~~
~~        b = Bunch(name="Python 3", language="Python 3.0.1")~~
~~        self.assertEqual("Python 3", b.name)~~
~~        self.assertEqual("Python 3.0.1", b.language)~~

if __name__ == "__main__":
    import unittest
    class TestBunch(unittest.TestCase):
        def test_attributes(self):
            b = Bunch(name="Python 3", language="Python 3.0.1")
            self.assertEqual("Python 3", b.name)
            self.assertEqual("Python 3.0.1", b.language)

    unittest.main()
```

While this works, it is really rather simpler to put the testing code into an entirely separate module that does not cause additional work when testing is not required. So we'll undo these modifications in a minute to keep the code in the remaining examples as straightforward as possible.

Adding a Behavior to the Bunch Class

The bunch class is useful in handling incoming data, but what about sending it out? For example, what if we want to print all the data? A first approximation to that task could simply use **print(b.__dict__)**, but the output is hardly user-friendly. You can easily add a method to the Bunch class.

CODE TO EDIT: bunchclass.py

```
"""
Simple bunch class with a pretty printing method
"""
import unittest

class Bunch(object):
    def __init__(self, *args, **kwargs):
        self.__dict__.update(kwargs)

    def pretty(self):
        text = ""
        for key, value in self.__dict__.items():
            text += "%s: %s\n" % (key, value)
        return text

class TestBunch(unittest.TestCase):
    def test_attributes(self):
        b = Bunch(name="Python 3", language="Python 3.0.1")
        self.assertEqual("Python 3", b.name)
        self.assertEqual("Python 3.0.1", b.language)
    def test_pretty(self):
        b = Bunch(name="Steve Holden", profession="Pythonista")
        p = b.pretty()
        self.assertTrue("name: Steve Holden" in p)
        self.assertTrue("profession: Pythonista" in p)
        self.assertEqual(len(p.splitlines()), 2, "Too many lines in output")

if __name__ == "__main__":
    ~~import unittest;~~
~~    class TestBunch(unittest.TestCase):~~
~~        def test_attributes(self):~~
~~            b = Bunch(name="Python 3", language="Python 3.0.1")~~
~~            self.assertEqual("Python 3", b.name)~~
~~            self.assertEqual("Python 3.0.1", b.language)~~
    unittest.main()
```

This version of the bunch class uses a **pretty()** method to display the attributes by accessing the object's magic **__dict__** property. Calling this method renders the attributes of the instance—with each key, value pair printing as **key: value**. Of course adding a new method means adding tests for it too, so **test_pretty()** tries to verify that it is creating the expected output.

Run your tests again to verify that they both succeed:

OBSERVE: the new test should pass first time

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Fixing a Bunch Class Issue

There is an issue with the updated Bunch class. It hasn't created any problems so far, but some interactive commands will make it clear:

CODE TO TYPE: Run the following code in an interactive interpreter session

```
>>> from bunchclass import Bunch
>>> b = Bunch(name="Audrey", job="Software Developer", pretty=True)
>>> b.pretty()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'bool' object is not callable
>>>
```

When we tried to call the **b.pretty()** method, we got a **TypeError** exception. This is because the argument **pretty=True** just overrode the **pretty()** method (remember: the interpreter looks for attributes in the instance's __dict__ before it looks in the class's __dict__), so the instance's **pretty** attribute is *masking* the class's **pretty()** method—the interpreter never gets around to looking in the class because it finds what it is looking for in the instance.

One solution is to use the built-in **hasattr()** and **setattr()** functions. Modify **bunchclass.py** to disallow masking of class attributes:

CODE TO EDIT: bunchclass.py

```
"""
Simple bunch class with a pretty printing method that protects its API.
"""

import unittest

class Bunch(object):
    def __init__(self, *args, **kwargs):
        ~~self.__dict__.update(kwargs)~~
        for key, value in kwargs.items():
            if hasattr(self, key):
                raise AttributeError("API conflict: '%s' is part of the '%s' API
" % (key, self.__class__.__name__))
            else:
                setattr(self, key, value)
    def pretty(self):
        text = ""
        for key, value in self.__dict__.items():
            text += "%s: %s\n" % (key, value)
        return text

class TestBunch(unittest.TestCase):
    ~~def test_attributes(self):~~
        ~~b = Bunch(name="Python 3", language="Python 3.0.1")~~
        ~~self.assertEqual("Python 3", b.name)~~
        ~~self.assertEqual("Python 3.0.1", b.language)~~
    def test_pretty(self):
        self.assertRaises(AttributeError, Bunch, name="Audrey", job="Software De
veloper", pretty=True)
        b = Bunch(name="Audrey", job="Software Developer")
        p = b.pretty()
        self.assertTrue("Audrey" in p)
        self.assertFalse("pretty: True" in p)
        ~~b = Bunch(name="Steve Holden", profession="Pythonista")~~
        ~~p = b.pretty()~~
        ~~self.assertTrue("name: Steve Holden" in p)~~
        ~~self.assertTrue("profession: Pythonista" in p)~~
        ~~self.assertEqual(len(p.splitlines()), 2, "Too many lines in output")~~


if __name__ == "__main__":
    unittest.main()
```

Run this program and see how the tests pass. Now, let's take a closer look at some of the code to understand what's going on.

OBSERVE: The Bunch class

```
class Bunch(object):
    def __init__(self, *args, **kwargs):
        for key, value in kwargs.items():
            if hasattr(self, key):
                raise AttributeError("API conflict: '%s' is part of the '%s' API
" % (key, self.__class__.__name__))
            else:
                setattr(self, key, value)
    def pretty(self):
        text = ""
        for key, value in self.__dict__.items():
            text += "%s: %s\n" % (key, value)
        return text
```

The **__init__** method uses the kwargs **items()** method which it gets for being of type **dict** to pass an iterable of keys and values that are tested for presence in the **self** object via the **hasattr** built-in. If the attribute doesn't exist yet, the **setattr** built-in is used to add the attribute. If the attribute does exist, we raise an **AttributeError**, which is used to identify when attribute assignment or references fail.

Python gives you the power to change the attributes of class objects almost at will. This is because Python makes the assumption that you are a "consenting adult" and understand the ramifications of what you do. This may sound a bit intimidating but this sort of confidence in the people who want to use Python is a hallmark of the language and the community that surrounds the language.

Note Certain applications—such as the control of nuclear reactors, flight control systems, and the like—require the ability to reason about program structures as a part of integrity verification. Dynamic languages like Python would require analysis that is too complex to be practical at today's state of the art.

Application Programming Interfaces

Suppose you wrote some software that lets you calculate something important, such as the speed of small birds tasked with carrying objects in a basket. This program would let an individual add and remove objects for the bird to carry, and when unladen it would simply go faster. You want to share this software with others, and to encourage its use, you want to make it easy for them to use.

The way this is done is through an Application Programming Interface, or **API**. An API refers to a specified interface between components, and allows us to build applications that use already existing software (and sometimes hardware). Sophisticated APIs drive our modern world and make it possible to send/receive email or text messages, take O'Reilly Software Courses, use on-line mapping tools, and a million other tasks. As a *designer*, your desire is to hide the complexity you have programmed into the classes from your users, who simply call API functions (and classes).

You will use what you learned in creating Bunch classes to build a simple API.

Designing an API

The first step in designing an API is to figure out what data it should handle and what behaviors it should have. Our API should have the following capabilities:

- **initialize:** Gives the user the ability to create a Bird object carrying any number of small objects in its basket.
- **add:** Add another object for the Bird to carry in its basket.
- **remove:** Remove an object from the Bird's basket
- **calculate:** Calculate the bird's current speed.
- **basket:** Return an attractive string that lists the materials in the basket.

Does this look very similar to how you have designed Python classes in the past? It should, because the preferred method in API design is to follow an object-oriented approach. This lets people find data represented by an object and then call behaviors and methods to act upon that data. Importing the class and creating instances automatically gives other programmers access to the API you have designed.

Building the API

We've got enough information to lay out the skeleton code for our API. Since the Bunch class already embodies a lot of the functionality we need, we'll subclass the Bunch class, allowing us to build on existing code. The bird API specification (which is what the following code essentially comprises) goes in a new file, **bird_api.py** in your **Python3_Lesson02/src** folder:

CODE TO TYPE: bird_api.py

```
"""
API for software birds carrying objects.
"""
from bunchclass import Bunch

class Bird(Bunch):

    def add(self, name, value):
        """
        Add an object for the Bird to carry in its basket.
            Name is a string naming the object
            Value is the actual object being placed in the basket.
        """

    def remove(self, name):
        """
        Remove an object from the basket
            name is the string of the object to be removed
        """

    def calculate(self):
        """
        Calculate the speed of the bird.
            algorithm: 100 - (5*number of objects in the basket)
            result cannot be less than zero.
        """

    def basket(self):
        """
        Print in an attractive format the list of objects in the basket.
        """

if __name__ == "__main__":
    swallow = Bird(fruit=("coconut", "orange"), drink="apple juice")
    swallow.add("cars", 3)
    print(swallow.basket())
    print(swallow.calculate())
    swallow.remove("drink")
    print(swallow.basket())
    print(swallow.calculate())
    help(swallow)
```

▶ Save and run it. You'll get nothing as a result except a bunch of **Nones** and the output from the **help()**. The help output is really critical because it allows you to view your software through the eyes of another programmer. This other guy or girl doesn't know any of the great stuff about your software that you do, so they will likely read the help to find out how to use your software—and whether they might want to. APIs without quality documentation are functionally impossible to use. Also, writing the documentation in an API you are providing can help you clean up the design.

If you have defined a module correctly, everything important in it should be documented. You should be able to verify this from the console window after running bird_api. Alternatively you can access the help from an interactive console session. (You may want to maximize the interactive console pane: select the console tab Console and then its □ maximize button—remember you can get back to your regular class view by selecting **Python** from the red leaf drop-down menu in Ellipse).

CODE TO TYPE: Accessing the bird API documentation interactively

```
>>> import bird_api
>>> help(bird_api.Bird)
Help on class Bird in module bird_api:

class Bird(bunchclass.Bunch)
 |  Method resolution order:
 |      Bird
 |      bunchclass.Bunch
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  add(self, name, value)
 |      Add an object for the Bird to carry in its basket.
 |          Name is what you call the object
 |      Value is the actual object being placed in the basket.
 |
 |  basket(self)
 |      Print in an attractive format the list of objects in the basket.
 |
 |  calculate(self)
 |      Calculate the speed of the bird.
 |          algorithm: 100 - (5*number of objects in the basket)
 |          result cannot be less than zero.
 |
 |  remove(self, name)
 |      Remove an object from the basket
 |          Name is the string of the object to be removed
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from bunchclass.Bunch:
 |
 |  __init__(self, *args, **kwargs)
 |
 |  pretty(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from bunchclass.Bunch:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Note that there is no need for individual method descriptions anywhere except in the docstring for that method. The help system collects all that information together for the user in one convenient place.

Now that our API is documented, let's add the behavior code in each method:

CODE TO EDIT: bird_api.py

```
"""
API for software birds carrying objects.
"""
from bunchclass import Bunch

class Bird(Bunch):

    def add(self, name, value):
        """
        Add an object for the Bird to carry in its basket.
            Name is what you call the object
        Value is the actual object being placed in the basket.
        """
        if hasattr(self, name):
            raise KeyError("'%s' object cannot be placed in basket")
        else:
            setattr(self, name, value)

    def remove(self, name):
        """
        Remove an object from the basket
            Name is the string of the object to be removed
        """
        if name in self.__dict__:
            delattr(self, name)
        else:
            raise KeyError("'%s' object not found in basket")

    def calculate(self):
        """
        Calculate the speed of the bird.
            algorithm: 100 - (number of objects in the basket * 10) minimum of 0
            result cannot be less than zero.
        """
        return max(100 - len(self.__dict__) * 10, 0)


    def basket(self):
        """
        Print in an attractive format the list of objects in the basket.
        """
        return "Basket Objects\n" + self.pretty()


if __name__ == "__main__":
    swallow = Bird(fruit=("coconut", "orange"), drink="apple juice")
    swallow.add("cars", 3)
    print(swallow.basket())
    print(swallow.calculate())
    swallow.remove("drink")
    print(swallow.basket())
    print(swallow.calculate())
    help(swallow)
```

Save and run it. We now have a working class that stores data in a structured format and has assigned behaviors to it. The documentation is such that you can easily figure out what is going on, making it possible to call it from other programs for a variety of uses. Suppose someone needed to model a flock of birds carrying objects from one location to another (perhaps they plan to start a courier service based on bird power).

Sharing APIs

A good aphorism for API work is "A good API is simply code, and code that is documented to the best of your ability." No one is going to want to use your API if all you do is provide a list of methods that *you* think is intuitive. Accurate and complete documentation is a hallmark of successful API design. Python's docstrings are a great tool for sharing your code.

There are a number of common ways to share an API. While outside the scope of this class, one of the more accessible methods is via the Internet. A very current example is the ability of social networking sites to provide cross-site login controls via an API called OpenID (http://en.wikipedia.org/wiki/OpenID). These more sophisticated APIs require the use of various modules taken from the Python standard library such as **urllib**, **ftplib**, **smtplib**, and more.

However, an API can also be called via simple object instantiation inside of code. In fact, this is the first method of testing done against an API during design. The Unit Tests with which you are familiar are commonly used in testing API designs and code.

Calling the API

In this section, we'll just call the API by importing the **Bird** class into a new program and using it. This is the most common way of using an API, and builds on what you already know. Create a new program named **flock** in your **Python3_Lesson02/src** folder as shown:

CODE TO TYPE: flock.py

```
from bird_api import Bird

class Flock(object):

    birds = []

    def add_bird(self, bird):
        """
        Add a bird object to the flock
        """
        self.birds.append(bird)

    def race(self):
        """
        Show how far the birds of the flock can go in one hour carrying their respective loads.
        """
        print("Distance flown in one hour by the flock")
        for bird in self.birds:
            distance = "-" * (bird.calculate() // 10)
            notice = "%s: %s carrying %s items" % (distance, bird.name, len(bird.__dict__))
            print(notice)

if __name__ == "__main__":
    swallow = Bird(coconut=1, name="Swallow")
    african = Bird(coconut=1, piece="of string", visited=False, name="African Swallow")
    european = Bird(coconut=1, lottery_numbers=(23, 12, 34), piece="of string", visited=True, name="European Swallow")
    european.add("cereal_boxes", 5)
    european.add("Norway", True)
    european.add("England", True)

    flock = Flock()
    flock.add_bird(swallow)
    flock.add_bird(african)
    flock.add_bird(european)
    flock.race()
```

In this API example, we import the **Bird** class from the **bird_api** module and call it to create a number of birds (**Bird** instances). We add the birds to our flock object and race them against each other. When you run the program the output clearly shows that the least-heavily laden swallow travels farthest.

OBSERVE: The output from running flock.py as a main program

```
Distance flown in one hour by the flock
--------: Swallow carrying 2 items
------: African Swallow carrying 4 items
--: European Swallow carrying 8 items
```

We use the API objects without modification, and only add attributes via the specified methods of the API. This is really important because Python is a very dynamic language that allows you many freedoms. You can break the API by replacing critical methods "from outside," as we found earlier in the case of the simple Bunch class.

There, the code triggered a **TypeError** exception because the **Bunch** class's **pretty()** method was *masked* by a data attribute on the instance, which we then attempted to call. While the Bunch class now protects this from happening during object instantiation, there is nothing to prevent you from masking the **pretty()** method simply by setting **bunch.pretty = True** after creating an instance.

Therefore, when using an object or value returned by an API, it is a good practice to use only the object's methods to modify its data (unless the documentation specifically gives you leave to change attribute values). To add further data, incorporate the objects into some other structure containing the associated information, as in the modification to **flock.py** shown below:

CODE TO EDIT: flock.py

```
from bird_api import Bird

class Flock(object):

    birds = []

    def add_bird(self, bird):
        """
        Add a bird object to the flock
        """
        self.birds.append(bird)

    def race(self):
        """
        Show how far the birds of the flock can go in one hour carrying their respective loads.
        """
        print("Distance flown in one hour by the flock")
        for bird in self.birds:
            distance = "-" * (bird.calculate() // 10)
            notice = "%s: %s carrying %s items" % (distance, bird.name, len(bird.__dict__))
            print(notice)
if __name__ == "__main__":
    swallow = Bird(coconut=1, name="Swallow")
    african = Bird(coconut=1, piece="of string", visited=False, name="African Swallow")
    european = Bird(coconut=1, lottery_numbers=(23, 12, 34), piece="of string", visited=True, name="European Swallow")
    ~~european.add("cereal_boxes", 5)~~
    ~~european.add("Norway", True)~~
    ~~european.add("England", True)~~

    birds = (
        ("Swallows are a group of birds in the family Hirundinidae.", swallow),
        ("African swallows are said to be able to carry coconuts.", african),
        ("European swallows are said to have trouble carrying coconuts.", european),
    )

    flock = Flock()
    ~~flock.add_bird(swallow)~~
    ~~flock.add_bird(african)~~
    ~~flock.add_bird(european)~~
    for stmt, bird in birds:
        print(stmt)
        flock.add_bird(bird)
    print("*"*40)
    flock.race()
```

Save and run it. You should see something like this:

OBSERVE: Output from the modified flock.py

```
Swallows are a group of birds in the family Hirundinidae.
African swallows are said to be able to carry coconuts.
European swallows are said to have trouble carrying coconuts.
****************************************
Distance flown in one hour by the flock
--------: Swallow carrying 2 items
------: African Swallow carrying 4 items
-----: European Swallow carrying 5 items
```

In this example, you used *tuples* to store some extra information along with the new bird objects. You didn't modify the existing API objects and so could be secure that the results would not throw an exception. Tuples are a valuable way to save associated data, since you know that no other portion of the code can modify the tuple because of its immutable nature.

Method Resolution Order

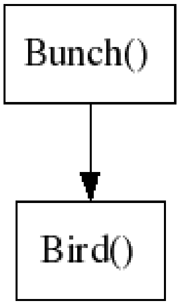
Let's think about a small family. For the sake of brevity we'll use "parent" instead of "mother" or "father," and "child" instead of "son" or "daughter." What we have then is a family consisting of a parent and child. The parent has certain features such as hair color and voice. The child when grown will have similar features to those of its parent. However, children generally have more than one parent, and their parents have parents, and so on. Determining the features the child inherits becomes complicated very rapidly—more so as you add in the unpredictability of genetics. Also, the child can modify their appearance and voice. Maybe they dye their hair or scream too much at concerts and their voice is altered. Now they have some features different from any of their ancestors.

In programming, names of familial relationships are used to describe similar relationships among object classes. Inheritance, parent, and child are frequently used to describe the elements of object inheritance. One noticeable difference in terminology is that instead of "features," object inheritance tracks the behaviors we call methods.

Another important difference is that programmatic inheritance does not have any genetic variety, instead being fixed and static. Programmers tend to prefer this: while life may lack interest without the rich profusion of genetic mutation, it does have a certain predictability which is welcome when thinking about what is actually happening in a program.

Basic Method Resolution Order

If you explore some of the previous code in this lesson, you can see the inheritance relationship between the Bunch and Bird classes:



Bunch is the parent of Bird, and Bird is the child of Bunch. Bird has all the methods of Bunch. We call the Bunch **__init__()** method when we instantiate a Bird object and the **pretty()** methods when we test the results of the Bird class.

Let's change the Bird's **pretty()** method. Bird is vain so instead of displaying its attributes we'll have the **pretty()** method return "**pretty bird**". In order to do this, all we need to do is add a new **pretty()** method to the Bird class to override what it inherited from the Bunch class:

CODE TO EDIT: bird_api.py

```
"""
API for software birds carrying objects.
"""
from bunchclass import Bunch

class Bird(Bunch):

    def pretty(self):
        """
        Replacement pretty() method
        """
        return "pretty bird!"

    def add(self, name, value):
        """
        Add an object for the Bird to carry in its basket.
            Name is what you call the object
        Value is the actual object being placed in the basket.
        """
        if hasattr(self, name):
            raise KeyError("'%s' object cannot be placed in basket")
        else:
            setattr(self, name, value)

    def remove(self, name):
        """
        Remove an object from the basket
            Name is the string of the object to be removed
        """
        if name in self.__dict__:
            delattr(self, name)
        else:
            raise KeyError("'%s' object not found in basket")

    def calculate(self):
        """
        Calculate the speed of the bird.
            algorithm: 100 - (number of objects in the basket * 10) minimum of 0
            result cannot be less than zero.
        """
        return max(100 - len(self.__dict__) * 10, 0)


    def basket(self):
        """
        Print in an attractive format the list of objects in the basket.
        """
        return "Basket Objects\n" + self.pretty()


if __name__ == "__main__":
    swallow = Bird(fruit=("coconut", "orange"), drink="apple juice")
    swallow.add("cars", 3)
    print(swallow.basket())
    print(swallow.calculate())
    swallow.remove("drink")
    print(swallow.basket())
    print(swallow.calculate())
    ~~help(swallow)~~
```

Save and run it. Instead of "fruit: ('coconut', 'orange')" you'll get "pretty bird!".

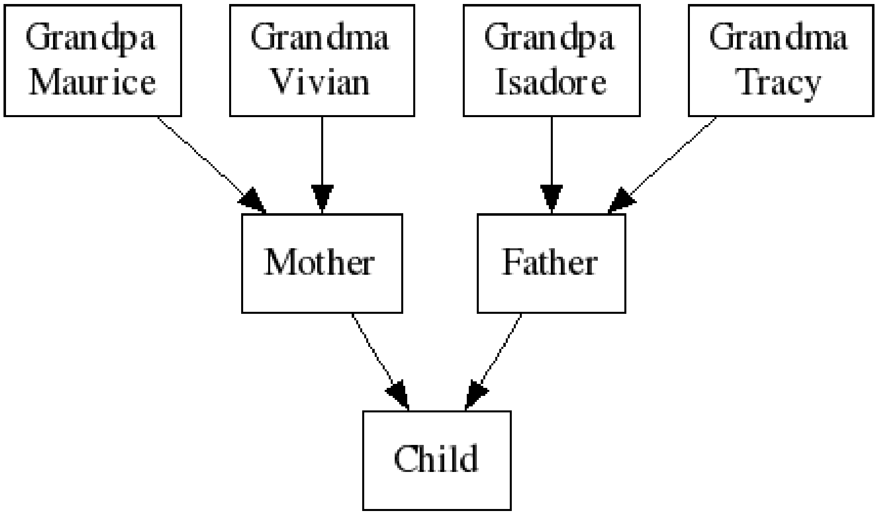
OBSERVE: Output from modified bird_api tests

```
Basket Objects
pretty bird!
70
Basket Objects
pretty bird!
80
```

To summarize, if a child class inherited a method from its parent class, you can override that inherited method by adding a method of the same name to the child class.

More Complicated Method Resolution Order

Let's continue with the family analogy. Regardless of the marital status, the child has two immediate genetic donors known by the common vernacular as "mother" and "father," or collectively as "parents." Those parents have parents of their own. Python lets you model this sort of genetic structure (and many others). With that in mind, let's model the hair color of the following inheritance structure:



Let's assume that we all have four different grandparents with four different hair colors—except Grandpa Isadore, who went bald early. Python gives precedence to the leftmost inherited object; to see what the child ends up with, create **inhairitance.py** and **test_inhairitance.py** in your **Python3_Lesson02/src** folder as shown:

CODE TO TYPE: inhairitance.py

```
"""
Complex inheritance program
"""

import unittest

class Maurice(object):
    def hair(self):
        return "red"

class Vivian(object):
    def hair(self):
        return "brown"

class Isadore(object):
    def hair(self):
        return "bald"

class Tracy(object):
    def hair(self):
        return "gray"

class Mother(Maurice, Vivian):
    pass

class Father(Isadore, Tracy):
    pass

class Child(Father, Mother):
    pass

if __name__ == "__main__":
    child = Child()
    print(child.hair())
```

CODE TO TYPE: test_inhairitance.py

```
"""
Inheritance test program
"""

import unittest
from inhairitance import Child

class TestHair(unittest.TestCase):

    def test_hair(self):
        child = Child()
        hair = child.hair()
        self.assertNotEqual(hair, "red")
        self.assertNotEqual(hair, "brown")
        self.assertNotEqual(hair, "gray")
        self.assertEqual(hair, "bald")

if __name__ == "__main__":
    unittest.main()
```

Save and run it. We can deduce by the fact that the tests succeed that the child's hair is "bald." This is because the method resolution order tries the "base classes" in its search for a method from left to right (in the order they are given in the class statement). Thus a method for child will be sought first in Father, then in Mother. Father, of course, has no **hair()** method, and so *its* base classes are searched, again in left-right order. This is known as a *left-first depth-first search*.

If you switch the order of inheritance, say, Mother with Father, the child will have red hair. Indeed, if genetics were as straightforward as programming in Python, a lot more people would be happy with their hair—except maybe Grandpa Isadore.

In any case, most of the time when you program, all you need is single inheritance and method resolution order in Python is pretty straightforward. Inheritance is a powerful tool but it can get complicated rather quickly. It is a good practice to keep inheritance as simple as possible with clearly named classes. If things get too complicated for simple inheritance, there are other tools you can wield from the programming armory to solve those problems.

Introspecting Inheritance Relationships

Python has two built-in functions that can help you in determining whether your code has been provided with values of a particular type. Note that this should not be a frequent requirement of your code, but it is sometimes justifiable usage.

issubclass(cls, classinfo) returns True if the object passed as **cls** is a direct or indirect subclass of one of the classes specified by **classinfo**. This second argument can either be a single class or a tuple of classes. In the latter case the result is True if **cls** is a subclass of any of the classes in the tuple. An indirect subclass of a class is a subclass of the class or one if its subclasses. For the purposes of **issubclass()** all classes are regarded as subclasses of themselves.

isinstance(obj, classinfo) returns True if the **obj** argument is an instance of some class that is a subclass of one of the classes specified by the **classinfo** argument. Again this argument may be either a single class or a tuple of classes.

Laying the Foundation

In this lesson, you've learned about some basic practices of structuring data in Python. We'll use these practices in further lessons; they are commonly used in real-world applications. Good data structuring takes practice and there are different standards on how to do it. The best methods result in code that is clear to read and easy to extend. The poor methods make code hard to interpret and "fragile"—the code often breaks without much warning. If you lay out a structure and it becomes hard for you to follow, often that means you need to stop and refactor your code. Fortunately, you wrote unit tests, right? If so, you can be reasonably confident that your refactoring has not caused any new defects.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Iteration in Python

This lesson includes the following sections:

- Iterables vs. Iterators
- Generators: Avoiding Creation of Large Sequences
- Generator Expressions

Iterables vs. Iterators

In the broadest possible terms, an iterable is something you can iterate over, and an iterator is what the interpreter uses to do the iteration. This description, however, is too general to be of enough value. The fundamental question is: what does the interpreter do when you write **for i in s:** in your program or other module?

In modern Python, iteration is supported by two quite separate mechanisms. So the answer to the question "how does the interpreter iterate over objects?" depends on the presence of specific methods on the object. If the object has an **__iter__()** method, then it is iterable using the new-style iteration mechanism. Otherwise the interpreter looks for a **__getitem__()** method and, if it finds one, uses the old-style iteration mechanism. If neither method is present, the interpreter raises a TypeError exception because the object is not iterable.

Old-Style Iteration

If an object, *o*, has no **__iter__()** method and you tell the interpreter to iterate over it, the interpreter initializes an internal variable to zero and repeatedly calls the object's **__getitem__()** method with successively higher values of the internal variable. From the point of view of the object, it's as though it were being manipulated by this code:

OBSERVE: Effective logic of an old-style for loop

```
# Approximate equivalent of:
#    for val in o:
#        # [loop body]
intern = 0
while True:
    try:
        val = o[intern]
    except IndexError:
        break
    # [loop body]
    intern += 1
```

In fact, you can create your own classes whose instances can be iterated over in this way. All you need to do is provide a **__getitem__(n)** method that raises an **IndexError** exception when the value of *n* is too high. Suppose you wanted to implement fixed-length sequences of objects. You could define a function to create an appropriate sequence (list or tuple or string) with the required number of components in it (so **fls("*", 12)** would return **"************"**, for example).

Alternatively, you could define an **fls** class, whose **__init__()** method had the same signature as the function above. Create a new **Python3_Lesson03** project and assign it to your **Python3_Lessons** working set. Then, create **fls.py** in the **Python3_Lesson03** project as shown.

CODE TO TYPE: fls.py

```
"""
Simple demonstration of the "old iteration protocol" - still available.
"""
class fls(object):
    def __init__(self, val, times):
        self.val = val
        self.count = times
    def __getitem__(self, n):
        if n >= self.count:
            raise IndexError("Object has no item %s % n")
        return self.val

thing = fls("*", 5)
for c in thing:
    print(c)

thing = fls(120,3)
for c in thing:
    print(c)
```

Save and run it. You see the following output:

OBSERVE: Output from running the fls class

```
*
*
*
*
*
120
120
120
```

So, for iteration purposes, you can see that the **fls** objects appear to act like other sequences, only with very boring behavior because all elements are constrained to be the same—the only value that **__getitem__()** ever returns is the one that was passed in to **__init__()**. But the main point is that you know a little more about Python's iteration mechanism. Now try a few other cases for yourself—use an interactive console session to create and test out some further **fls** objects interactively.

Note Remember you will need to import the fls class from the fls module in order to be able to create instances of it.

New-Style Iteration

The iteration mechanism outlined above is all very well when you are iterating over numbered items in a sequence, but it does not naturally extend to collections like sets and dicts, which do not specify a natural ordering for their items. Dicts, in fact, do have a **__getitem__()** method, but it takes a key value and returns the appropriate item (assuming that a key with that value exists—if there is no such key, it raises a **KeyError** exception). Sets don't even *have* a **__getitem__()** method, since they are effectively "item-less dicts".

It was to overcome issues like this that the "new-style" iteration protocol was defined. You learned above that the interpreter will look for an **__iter__()** method on the objects that you iterate over. If it *finds* **__iter__()**, it uses it to create an *iterator* from the *iterable* you are iterating over.

The iterator will have a **__next__()** method—this is a requirement of the iteration protocol. Each time around the loop, the interpreter obtains the next value for the iterable by calling the iterator's **__next__()** method. Again, you can perhaps understand this more easily with an approximate Python equivalent to a for-loop over a new-style iterable:

OBSERVE: Effective logic of an new-style for loop

```
# Approximate equivalent of:
#    for val in o:
#        # [loop body]
it = o.__iter__()
while True:
    try:
        val = it.__next__()
    except StopIteration:
        break
    # [loop body]
```

You may wonder why Python insists on creating a new object for each iteration: couldn't it just use the iterable directly? The answer to that question is "no": the iterator contains the current state of the iteration, and code that iterates over the same iterable twice is perfectly legal. Iterating over the same *iterator*, however, gives results that are not usually what you want. You can see this by playing with the interactive interpreter.

CODE TO TYPE: Enter the following in an interactive interpreter session

```
>>> lst = [1, 2, 3]
>>> dir(lst)
[..., '__getitem__', ..., '__init__', ...]
>>> for i in lst:
...     for j in lst:
...         print(i, j)
...
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
>>> li = lst.__iter__()
>>> dir(li)
[..., '__iter__', ..., '__next__', ...]
>>> for i in li:
...     for j in li:
...         print(i, j)
...
1 2
1 3
>>> lii = li.__iter__()
>>> li
<list_iterator object at 0x01995270>
>>> lii
<list_iterator object at 0x01995270>
>>> l2 = lst.__iter__()
>>> l2.__next__()
1
>>> l2.__next__()
2
>>> l2.__next__()
3
>>> l2.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

What's the difference between the first **for** loop and the second? Look at the **dir()** listing of lst, which is a list instance, and note that it has both **__iter__()** and **__getitem__()** methods. When the interpreter iterates over the list, calling its **__iter__()** method creates a new iterator, yielding a complete sequence of values, every time it encounters a **for** loop.

We then created a list iterator object by manually calling our list's **__iter__()** method. Note that the list iterator object also has an **__iter__()** method, and adds a **__next__()** method, but it lacks a **__getitem__()**. The **__iter__()** method of the iterator is rather different from that of the list, however:

OBSERVE: __iter__() method - in Python it would read:

```
def __iter__(self):
    return self
```

In other words, each time you iterate over a list (which is an *iterable*), the call to its **__iter__()** method creates a new iterator, which has its own independent state. The iterator's **__iter__()** method, however, does not create a new iterator, which means that the inner and outer loops are sharing the same iterator. This in turn means that by the time the outer loop is trying to begin its second iteration, the iterator has already been exhausted by the inner loop and (for the second time) raises the **StopIteration** exception.

The final few statements demonstrated this by manually going through the steps that the interpreter does when iterating over a list. We saw the l2 iterator produce three values on successive **__next__()** calls before raising a **StopIteration** exception. Normally, of course, the exception is caught internally by the logic of the **for** loop, and therefore does not become visible.

In summary, calling an iterable's **__iter__()** method creates an iterator that can be used to iterate over the iterable.

Creating Your Own Iterators

Now that you understand Python's iteration processes somewhat better, you may be wondering whether you can define your own iterable classes. The answer is "yes"! You will need to provide an **__iter__()** method (which can simply return **self** if you are implementing an iterator rather than a more general iterable: this is usually OK, since when you write an iterator class it is easy to create multiple instances, each having independent state). The **__next__()** method should return successive values until there are no more, at which point it should raise a **StopIteration** exception.

Rather than create an example now, we'll create it in the next section. First, we'll create a generator, and then we'll build an equivalent iterator.

Generators: Avoiding Creation of Large Sequences

The iteration protocol discussed above also comes into play with so-called *generator functions*. The only apparent difference between a generator function and the regular kind you have dealt with before is the appearance of the **yield** keyword in the function body. So what's the difference between a regular function and a generator function?

The answer is that calling a generator function produces a special type of iterator object (a "generator"). The function namespace is created and initialized with the argument values. The function code only starts executing with the first call to the generator's **__next__()** method. Execution continues until a **yield** expression is evaluated: the value of the expression following **yield** becomes the value of the **__next__()** method call. You can see this with a very simple generator function in an interactive session.

CODE TO TYPE: Enter the following code in an interactive interpreter window

```
>>> def g(x):
...     yield x
...     x *= 2
...     yield x
...
>>> g
<function g at 0x02286A98>
>>> gen = g("##")
>>> gen
<generator object g at 0x02285A08>
>>> dir(gen)
['__class__', ..., '__iter__', ..., '__next__', ...]
>>> gen.__next__()
'##'
>>> gen.__next__()
'####'
>>> gen.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> gen.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

g() is a generator function, though when you ask the interpreter about it you don't see any difference from any other function. Calling it creates a generator object, though, and the **dir()** listing shows that it has the necessary methods for an iterator. Calling the object's **__next__()** method returns the result of the next **yield** expression in the function's code body.

If the function ends before encountering a **yield** expression (either by executing a **return** statement or dropping off the bottom), the **__next__()** method call raises a **StopIteration** exception just like any other iterator. Also note that, once the generator starts to raise **StopIteration** exceptions when **__next__()** is called, it continues to do so for each subsequent call—the iterator is exhausted.

Advantages of Generator Functions

The really convenient thing about generator functions is that they allow you to perform all sorts of complex calculations to produce the values in a sequence, but the code that consumes (makes use of) these values can be entirely separated from the generator that produces them. The values are consumed in a simple **for** loop—or any other similar iterative context in Python, such as a list comprehension.

Not only do they make your code simpler by separating out the production and consumption of sequences, but generators allow you to create sequence values one at a time, as they are consumed. There is no need to build a list or tuple to store them in, which means your programs will use less storage and operate more quickly (though these advantages do not really make much difference unless the number of objects becomes large).

A Simple Generator Function

Suppose you need to produce sequences determined by a list, but need to repeat the first list element once, the second twice, and so on. So given a list **[2, 4, 6]**, the resulting sequence would be 2, 4, 4, 6, 6, 6. Let's write a generator that produces such sequences. First, though, we'll write tests to ensure that our generator function works. Create **testgen.py** in your **Python3_Lesson03/src** folder as shown:

CODE TO TYPE: testgen.py

```
"""
testgen.py: simple test for a sequence generator
"""
import unittest
from gen123 import gen123

class TestGen(unittest.TestCase):

    def testEmpty(self):
        self.assertEqual(list(gen123([])), [], "Empty list does not give empty l
ist")

    def test123(self):
        self.assertEqual(list(gen123([1])), [1], "[1] does not give [1]")
        self.assertEqual(list(gen123([1, 2])), [1, 2, 2])
        self.assertEqual(list(gen123([1, 2, 3])), [1, 2, 2, 3, 3, 3])

if __name__ == "__main__":
    unittest.main()
```

As usual, we start out with a simple stub function to make sure that the tests fail. Now, create **gen123.py** in your **Python3_Lesson03/src** folder as shown:

CODE TO TYPE: gen123.py

```
"""
gen123.py: generate sequences from a base list, repeating
           each element one more time than the last
"""

def gen123(m):
    yield None
```

Save and run the test program:

OBSERVE: Results from running testgen.py

```
FF
======================================================================
FAIL: test123 (__main__.TestGen)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson03\src\testgen.py", line 13, in test123
    self.assertEqual(list(gen123([1])), [1], "[1] does not give [1]")
AssertionError: [1] does not give [1]

======================================================================
FAIL: testEmpty (__main__.TestGen)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson03\src\testgen.py", line 10, in testEmpty
    self.assertEqual(list(gen123([])), [], "Empty list does not give empty list"
)
AssertionError: Empty list does not give empty list

----------------------------------------------------------------------
Ran 2 tests in 0.032s

FAILED (failures=2)
```

Now, let's see how it does with some real code in there.

CODE TO EDIT: gen123.py

```
"""
gen123.py: generate sequences from a base list, repeating
           each element one more time than the last
"""

def gen123(m):
    ~~yield None~~
    n = 0
    for item in m:
        n += 1
        for i in range(n):
            yield item
```

Save and run the test program:

OBSERVE: Output from testgen.py; the tests now pass

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

An Iterator Equivalent of the Generator

As you learned above, it is also possible to write classes that obey the iteration protocol. You will end this lesson by writing an iterator equivalent of the generator function above. Since you want it to perform exactly the same as the gen123 generators, you can use the same tests to verify its operation—that is one of the benefits of a test-driven environment! The new component should ideally be a "drop-in replacement" for the generator function. Create **class123.py** in your **Python3_Lesson03/src** folder as shown:

CODE TO TYPE: class123.py

```
"""
A simple iterator object specification.
"""
class gen123:
    def __init__(self, lst):
        "Initialize the iterator object."
        self.lst = lst
        self.itemno = 0
        self.count = 1
    def __iter__(self):
        "This object is not an iterable."
        return self
    def __next__(self):
        "Return the next value in the output sequence."
        if self.count > self.itemno:
            try:
                self.val = self.lst[self.itemno]
            except IndexError:
                raise StopIteration
            self.itemno += 1
            self.count = 1
        self.count += 1
        return self.val
```

This code is considerably more complex. This should not be surprising, because generator functions were devised to solve this type of problem cleanly and simply.

Instead of calling the generator function, the test routine will now call your iterator's class (which, you will notice, has the same name). This causes its **__init__()** method to be run, and the list of values is stored as an instance variable. Two other instance variables are initialized: one to keep track of which item is currently being output, and the other to keep track of how many times the current value has been produced.

All the magic, of course, takes place in the **__next__()** method. First it checks to see whether it is time to move to the next element of the value list (the item number and count are set up initially to ensure that this branch is actioned on the first call). If so, the **val** instance variable is retrieved.

If no more values are available, the method raises a **StopIteration** exception to terminate the loop. Note carefully that this action can be repeated—once the method starts to raise the exception, it should be raised for every subsequent call.

Once the correct item value is established, the count is incremented and the value is returned as the result of the call.

This code is about twice as long as that of the generator solution, and so you would probably choose to write a generator function for problems like this. But if you need close control over iterative behavior, you may end up needing to write your own iterators.

Testing the module is easy. Just make the following change to the test program:

CODE TO EDIT: testgen.py

```
"""
testgen.py: simple test for a list generator function
"""
import unittest
from class123 import gen123

class TestGen(unittest.TestCase):

    def testEmpty(self):
        self.assertEqual(list(gen123([])), [], "Empty list does not give empty l
ist")

    def test123(self):
        self.assertEqual(list(gen123([1])), [1], "[1] does not give [1]")
        self.assertEqual(list(gen123([1, 2])), [1, 2, 2])
        self.assertEqual(list(gen123([1, 2, 3])), [1, 2, 2, 3, 3, 3])

if __name__ == "__main__":
    unittest.main()
```

Save and run the updated test program; you should see a successful result immediately, thereby giving strong evidence that the two implementations are equivalent.

Generator Expressions

After the new-style iteration protocol was adopted in Python, one of the developers observed that it would be very useful to be able to write expressions that were similar to list comprehensions in using iteration (**for**) and selection (**if**) elements to produce expressions that generated their results rather than producing a list. The reasoning behind this is just the same as the reasoning behind standard generators—creating the objects one by one "on demand" is more space-efficient, and is likely to speed up programs dealing with large sequences considerably, as well as reducing their memory requirements.

The syntax of a generator expression is the same as for list comprehensions (learned in an earlier course), but with parentheses instead of brackets. Because they are generators, however, you only see the individual values when you consume them inside an iteration. Learn a little more about them by playing in an interactive interpreter session.

INTERACTIVE SESSION:

```
>>> gx1 = (x for x in range(10) if x % 3)
>>> gx1
<generator object <genexpr> at 0x0230FD78>
>>> list(gx1)
[1, 2, 4, 5, 7, 8]
>>> list(gx1)
[]
>>> sum(i for i in range(100))
4950
>>> gx2 = (ord(c) for c in "Jim")
>>> next(gx2)
74
>>> next(gx2)
105
>>> next(gx2)
109
>>> next(gx2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Note that the generator expressions are iterators, but not iterables: once you have iterated over them they are exhausted, and any further attempt to iterate over the expression raises an immediate **StopIteration**. Also observe that you used a new built-in function in that session. Calling **next(o)** is pretty much equivalent to calling **o.__next__()**, right down to the raising of a **StopIteration** execution when no more values are available.

Generators and generator expressions primarily offer memory savings, though this can equate to time savings if you are avoiding a lot of memory allocation and deallocation. For very large data sets, it can make a computation practical that you might otherwise not have enough memory for.

You now know much more about the way Python iterates over objects than you formerly did. With luck, this knowledge will allow you to build objects that help you solve your problems more effectively.

When you finish the lesson, return to the syllabus and complete the quizzes and projects.

Basic Regular Expressions

Suppose you've been given a big block of text and told you need to pull all of the US-style phone numbers from it. Writing a program like that requires breaking up all the words via the String **split()** method, then writing code to make sure that the numbers and dashes are all in the right places. We are talking about at least a dozen lines of code, and that doesn't even begin to account for special cases, like when the area code is in parentheses.

What if there was a special syntax so that you could find those numbers with a single line of code? Something like **xxx-xxx-xxxx or (xxx) xxx-xxxx** that you could apply to the text? The "x" would mean "any number," and the pattern would be applied and would return a list.

Fortunately, there is: Python lets you use *regular expressions*, which do that and much more besides! They aren't the answer to every string-related problem, but regular expressions are an important part of any developer's toolkit. This lesson will go over the basics of what you can do with regular expressions and will be followed by a more complete exposition of the capabilities of the **re** module.

In the 1950s, mathematician Stephen Cole Kleene described automata theory and formal language theory in a set of models using a notation called *regular sets* as a method to do pattern matching. Active usage of this system, called Regular Expressions, started in the 1960s and continued under such pioneers as David J. Farber, Ralph E. Griswold, Ivan P. Polonsky, Ken Thompson, and Henry Spencer.

Regular expressions, also called *re*s or *regex*es, provide a concise and flexible means for matching strings of text. They are a common programming tool used not just in Python but many languages in common use today.

This lesson includes these sections:

Matching and Searching

The **re** module provides features to enable pattern matching in Python. The basic mode of operation is to call either the **match()** or **search()** function from that module with a regex as the first argument, and a string to match against as the second argument. If the regex matches the string, the module returns a match object, and analysis of the match object can give you information about (for example) the exact strings matched by various portions of the pattern.

INTERACTIVE SESSION:

```
>>> import re
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m
<_sre.SRE_Match object at 0x01D557C8>
>>> m.groups()
('Isaac', 'Newton')
>>> m.group(0)
'Isaac Newton'
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
>>> m = re.match(r"(..)+", "a1b2c3")   # Matches 3 times.
>>> m.group(1)                         # Returns only the last match.
'c3'
>>> m = re.search("\\d+", "hello123extra")
>>> m.start()
5
>>> m.end()
8
>>> m.span()
(5, 8)
>>> m.group(0)
'123'
>>>
```

In the interactive session above, we used the **re** module's **match()** and **search()** functions to determine whether strings conformed to a specific pattern (provided as the first argument to the function call). **match()** requires the pattern to occur at the start of the target string, while **search()** will move through the target string looking for it. If the pattern is not present in the string, the function call returns None. Otherwise it returns a *match object* ("m," above) that can be queried for specific aspects of the matched string by calling its various methods.

Finding Characters: Regular Expression Patterns

A regular expression pattern is a way of describing a set of character strings. These descriptions can be relatively concise: the pattern "x" matches precisely one character, the lowercase letter "x." Some characters have special meanings, so the pattern "x+" matches any string of one or more lower case "x"s—the plus sign generates a more complex pattern from the pattern it follows.

Most characters can be used in patterns like the lower case "x" to "stand for themselves," so for example if you wanted to match the literal string "thing" you would do so with a pattern that reads "thing"—the "t" in the pattern matches a "t" in the string, and so on. But there are quite a few abbreviations: for example, '\d' matches any decimal digit (making it equivalent to the pattern "[0123456789]," as we will learn shortly). Here are some of the more common abbreviations.

Pattern String	Description
.	Matches any character except a newline in the target string.
^	Matches the start of the target string, or the start of a line within the target string.
$	Matches the end of the string, or just before the end of a line within the string. **foo** matches both "foo" and "foobar," while the regular expression **foo$** matches only "foo."
*****	Matches the regex it follows, zero or more times, so **ab*** will match "a," "ab," or "a" followed by any number of "b"s.
+	Matches the regex it follows, one or more times, so **ab+** will match "a" followed by any number of "b"s, but will not match "a" alone.
?	Optionally matches an occurrence of the regex that precedes it. **ab?** matches either of "a" or "ab."
	Matches special characters literally, allowing you to match plus signs, asterisks and other characters

\	having special significance in regexes. Also introduces a special sequence such as '\d' to match any digit.
{m} (where m is an integer)	Matches exactly *m* occurrences of the regex it follows.
{m,n} (where m and n are integers)	Matches between *m* and *n* occurrences of the regex it follows.
[...]	Matches any one of the set of characters appearing between the brackets. Special characters do not have their usual significance inside brackets, so **[abc$]** matches any of "a," "b," "c" or "$." A dash (-) between two characters specifies a range, so **[a-z]** matches any lower-case character.
[^...]	Matches any character *except* one of the set appearing after the caret between the brackets. Note that the caret only has this special meaning when it *immediately* follows the opening left bracket.
|	Alternation. **A|B**, where A and B are any regexes, first tries to match A and, if that fails, tries to match B. Any number of regexes can be used as alternates in this way, not just two.
(...)	Groups a number of regexes together, usually for the purpose of treating them as a single element (for example, to use as an alternate with |). When a match object is created, the string matched by the parenthesized group is available using methods of the match object.

Tip There are many regex references available on the Internet; you might want to find and bookmark one or two of them!

Grouping in Patterns

As the last line above indicates, patterns can contain *groups*, indicated by parentheses. The strings matched by the groups are, under certain circumstances, available—again, by calling the match object's methods. The groups can be numbered (according to their relative positions in the pattern, and starting at one rather than Python's usual zero—group 0 refers to the match as a whole) and they can also be named if the group's opening parenthesis in the pattern is followed by a question mark, an upper case "P" and a name in angle brackets, as we saw with "(?P<first_name>\w+)" in the earlier interactive session.

Groupings in the pattern are the principal way of extracting required information from the match object. Strings matched by non-grouping portions of the pattern cannot be individually identified in the match object. When you are testing a new regular expression, it is often useful to interactively inspect the result of calling the match objects' **groups()** and **groupdict()** methods to verify that your pattern is matching as you expect.

When testing patterns, you can test for equality with those objects; but you should also remember to test that unacceptable strings are *not*, in fact, matched. This will usually involve the use of your test case's **assertNone()** method on the match result.

Substitution for Patterns

Besides the **match()** and **search()** functions, the **re** module provides functions that allow you to make replacements of patterns in the target string (these functions return new strings, of course, because strings are immutable in Python). The **re.sub()** function takes not only a pattern and a target string but also a replacement string, as shown below.

re.sub() Syntax

```
re.sub(pat, replacement, target[, count, flags])
```

This replaces each non-overlapping occurrence of the given **pattern** in the **target** string with the **replacement** element given as the third argument. If **replacement** is a string, any backslash escapes in it are processed down to individual characters (so, for example, "\n" is replaced by a newline character). Escapes of the form *n* (where *n* is a decimal digit) allow replacement by one of the matched groups from the pattern. The **replacement** argument can also be a function, in which case it is called for each replacement with a single argument, which is the match object corresponding to the currently matched string that is to be replaced.

The optional argument **count** is the maximum number of pattern occurrences to be replaced; **count** must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous match, so sub('x*', '-', 'abc') returns '-a-b-c-'. The **flags**, if present, are the usual regular expression matching flags, which we'll discuss a little later. Let's get an idea of what you can do with the replacement facilities.

This example simply shows straight pattern replacement: both calls to re.sub() replace all occurrences of the string "1" with a newline character. When the pattern contains characters like newline (normally represented by escape sequences), or special patterns (which also require backslashes), r(aw) strings can make patterns more readable and easier to type. The more complex the patterns become, the truer this is.

INTERACTIVE SESSION:

```
>>> import re
>>> re.sub("1", "\\n", "123123123123") # replace digit one with newline
'\n23\n23\n23\n23'
>>> re.sub("1", r"\n", "123123123123") # replace digit one with newline
'\n23\n23\n23\n23'
```

The next call to **re.sub()** uses a function to supply the replacement string: if the function is replacing a single minus sign it returns a space, but two minus signs are translated into a plus sign.

INTERACTIVE SESSION:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == "-": return " "
...     else: return "+"
...
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro++gram files'
```

The next example finds the "#" marker and removes it and everything after it, then removes everything but the digits from the remaining string.

INTERACTIVE SESSION:

```
>>> s = "(123) 456-7890 # Commented phone number"
>>> nocomment = re.sub("#.*$", "", s)
>>> nocomment
'(123) 456-7890 '
>>> re.sub(r"\D", "", nocomment)
'1234567890'
```

These examples attempt to match any string beginning and ending in an at sign ("@") with zero or more sequences of "=+=" in the middle (the "+" must be escaped to make the matching code treat it as an ordinary character).

INTERACTIVE SESSION:

```
>>> re.sub("@(=\+=)*@", "xxx", "@@")
'xxx'
>>> re.sub("@(=\+=)*@", "xxx", "@=+=@")
'xxx'
>>> re.sub("@(=\+=)*@", "xxx", "@=+==+=@")
'xxx'
>>> re.sub("@(=\+=)*@", "xxx", "@=+=+=@")
'@=+=+=@'
```

The last example shows a pattern (a single vowel) being used to make many replacements—all vowels in the target string are replaced with a dash.

INTERACTIVE SESSION:

```
>>> re.sub("[aeiouAEIOU]", "-", "The Quick Brown Fox Jumps Over the Lazy Dog")
'Th- Q--ck Br-wn F-x J-mps -v-r th- L-zy D-g'
>>>
```

Trying Out Patterns

It's useful to be able to try out lots of patterns as you are learning how they are made up. See if you can understand the following patterns by trying them against various strings. To help you do that, we'll write a little program that allows you to see the results of searching and matching for a specific pattern against a number of strings. The program reads a pattern, and if it's the empty string, terminates. Otherwise, it reads target strings and applies matches and searches on the strings that are subsequently input until the user enters an empty string, in which case it goes back to requesting a new pattern. Create a **Python3_Lesson04** project and assign it to the **Python3_Lessons** working set. Then, create **pattest.py** in your **Python3_Lesson04/src** folder as shown:

CODE TO TYPE: pattest.py

```
"""
pattest.py: Allows the checking of various patterns and target strings
"""
import re
while True:
    pat = input("Pattern: ")
    if not pat:
        break
    while True:
        s = input("Target : ")
        if not s:
            break
        mm = re.match(pat, s)
        if mm:
            print("Match : matched {0!r}".format(s[mm.start():mm.end()]))
            print("Match : groups:", mm.groups())
            print("Match : gdict :", mm.groupdict())
        else:
            print("Match : no match")
        ms = re.search(pat, s)
        if ms:
            print("Search: matched {0!r}".format(s[ms.start():ms.end()]))
            print("Search: groups:", ms.groups())
            print("Search: gdict :", ms.groupdict())
        else:
            print("Search: no match")
```

This lets you test many strings against the same pattern quite quickly. Run it and ensure that you can think of strings that both match and don't match the patterns given below.

Pattern	Description
[0123456789]+	Matches one or more decimal digits.
[\d]+	Same as above. Remember to verify that some strings don't match the pattern -
[\w]+ +[\w]+	Matches two words separated by any number of spaces.
\(\d\d\d\) \d\d\d-\d\d\d\d	Matches a US telephone number with parentheses around the area code and a dash between the exchange and the number.
home-?brew	There should be exactly two strings that match this pattern.
\$\d+(\.\d{2})?	An amount of money (in dollars) with optional cents.

We've made a start on the use of regular expressions. While they aren't the answer to every problem, they can help to solve tricky text recognition problems. Just don't treat them as the first weapon in your arsenal—the string methods were provided for a reason! In the next lesson, we'll expand our knowledge of regular expressions further.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

More On Regular Expressions

This lesson includes these sections:

- Fundamentals of Regular Expressions
- Use Regular Expressions With Care

Fundamentals of Regular Expressions

Now that we've learned the basics of regular expressions, we can look at some more advanced aspects. Remember at the start of the last lesson, we introduced regular expressions by wondering how we might search for US telephone numbers in a specific text. You are now in a position to solve that problem.

The Telephone Number Search

We want to search a block of text for phone numbers in Python using Regular Expressions. As usual, first, we'll write a test to confirm that we're getting the behavior we want, and then we'll write the code. Create the **Python3_Lesson05** project and assign it to your **Python3_Lessons** working set. Then, in your **Python3_Lesson05/src** folder, create **test_phone.py** as shown:

CODE TO TYPE: test_phone.py

```
import unittest
from phone import get_phone, text

class TestRegex(unittest.TestCase):

    def test_phone(self):
        numbers = get_phone(text)
        self.assertEqual(len(numbers), 5)

if __name__ == "__main__":
    unittest.main()
```

Save the test program. Our first code finds only the phone numbers whose area code is not surrounded by parentheses, and the test is satisfied as long as the function detects five phone numbers in the text—without verifying that it has the exact numbers right. It is, however, *much* better than not having any tests!

phone.py

```
"""
Demonstrate use of re.findall().
"""
import re

text = """While I was at the store I tried to call 555-123-4567 on my mobile
but accidentally called 555-754-4321.  The person on the line redirected me to
999-999-9999 which I don't think is a real number. Neither is 000-000-0000 or 55
5-555-0000.
Well, I will try (555) 123-4567 again now.
"""

def get_phone(text):
    "Scan a text, locating telephone numbers."
    # Note the use of a "raw" string constant
    return re.findall(r"\d\d\d-\d\d\d-\d\d\d\d", text)

if __name__ == '__main__':
    print(get_phone(text))
```

Save and run phone.py.

OBSERVE: The output of phone.py

```
['555-123-4567', '555-754-4321', '999-999-9999', '000-000-0000', '555-555-0000']
```

Now, run test_phone.py.

OBSERVE: The output of the test

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Note In this lesson we'll use tests very heavily, because in regular expressions it can be easy to generate false positives: answers that return positive values but fail in some way.

phone.py

```
Demonstrate use of re.findall().
"""
import re

text = """While I was at the store I tried to call 555-123-4567 on my mobile
but accidentally called 555-754-4321.  The person on the line redirected me to
999-999-9999 which I don't think is a real number. Neither is 000-000-0000 or 55
5-555-0000.
Well, I will try (555) 123-4567 again now.
"""

def get_phone(text):
    "Scan a text, locating telephone numbers."
    # Note the use of a "raw" string constant
    return re.findall(r"\d\d\d-\d\d\d-\d\d\d\d", text)

if __name__ == '__main__':
    print(get_phone(text))
```

The first thing this program does is **import** the Python regular expression library, **re**. The **get_phone()** function uses a regular expression as the first argument to the **findall()** function from that library. The pattern **\d\d\d-\d\d\d-\d\d\d\d** is applied to the **text**, and the result is a list of strings matched by the pattern.

The regex pattern **"\d\d\d-\d\d\d-\d\d\d\d"** is the central component of the code above. If you replace each "**\d**" with an "X," you get **XXX-XXX-XXXX**—the template for matching the phone numbers.

You probably noticed that your code does not find the phone number with the prefix in parentheses. We'll cover that later in this lesson.

Regular Expressions and Raw Strings

Regular expressions often use the backslash (\) character. Mostly it indicates special meanings for the characters immediately following, but it can also be used to "escape" the standard meanings of certain characters in pattern strings, so that you can recognize these special characters too. You probably remember that the backslash also has a special meaning in string literals ("\n" means newline, "\t" means tab, and so on).

You probably remember that to represent a single backslash in a string, you normally need to use two backslashes—"\\." This would make regular expressions very difficult to read. Consequently, we have "raw" string constants (whose representations are preceded by the letter "r") to represent regex patterns. These let you represent the backslashes without escaping, which makes them much more readable.

match() vs search()

The basic use case for regular expressions is finding occurrences of strings that conform to a pattern. The

Python regular expression library gives you two ways to perform this action, **re.match()** and **re.search()**. The difference between them is as follows:

- **match()** checks at the start of a string and returns None if nothing is found.
- **search()** moves up the string, looking for the first occurrence of the given pattern, and returns None only if the pattern occurs nowhere in the string.

For example, suppose we have several paragraphs and want to see if they start with or contain a phone number. If a paragraph starts with a phone number, we'll assume that the paragraph is just a phone number and we want to return it. Otherwise, if a paragraph contains a phone number, we want to return the length of the paragraph. If a paragraph has no telephone numbers, we'll return **None**. Create **test_match_vs_search.py** in your **Python3_Lesson05/src** folder as shown:

test_match_vs_search.py

```
import unittest
from match_vs_search import check_number

p1 = """While I was at the store in Washington, DC 20001 I tried to call 555-123
-4567 on my mobile
but accidentally called 555-754-4321.  The person on the line redirected me to
999-999-9999 which I don't think is a real number. Neither is 000-000-0000 or 55
5-555-0000.
Well, I will try (555) 123-4567 again now."""

p2 = "555-555-5555"

p3 = "What is the author's phone number?"

class TestRegex(unittest.TestCase):

    def test_match(self):
        result = check_number(p2)
        self.assertEqual("555-555-5555", result)

    def test_search(self):
        result = check_number(p1)
        self.assertEqual(305, result)

    def test_none(self):
        result = check_number(p3)
        self.assertIsNone(result)

if __name__ == "__main__":
    unittest.main()
```

Save it and create **match_vs_search.py** in the same folder:

match_vs_search.py

```
"""
Demonstrate the difference between match() and search().
"""

import re

def check_number(text):
    regex = r"\d\d\d-\d\d\d-\d\d\d\d"
    match = re.match(regex, text)
    if match:
        return match.group()
    match = re.search(regex, text)
    if match:
        return len(text)
```

Save it and run the test program:

OBSERVE: Running test_match_vs_search.py

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Now, let's try our match_vs_search program.

CODE TO TYPE: Check the difference between match() and search()

```
>>> from match_vs_search import *
>>> check_number("707-867-5309")
'707-867-5309'
>>> check_number("Jenny's number is 707-867-5309")
30
```

Let's look at how it works.

OBSERVE: The check_number() Function

```
def check_number(text):
    regex = r"\d\d\d-\d\d\d-\d\d\d\d"
    match = re.match(regex, text)
    if match:
        return match.group()
    match = re.search(regex, text)
    if match:
        return len(text)
```

The **check_number()** function first attempts to match a telephone number at the beginning of the **text**. If that succeeds, it returns a *match object* **match**, described below.

If the **re.match()** call fails to find the pattern, it returns **None**, and the function then calls the **re.search()** function to try and find a number somewhere in the interior of the **text**. If the search succeeds, then the function returns the length of the paragraph. Otherwise it "falls off the bottom" and returns None (as is standard in Python).

Let's continue our session to explore the difference between match() and search():

CODE TO TYPE: Check the difference between match() and search()

```
>>> import re
>>> target = "This is a string"
>>> def t(p, t):
...     if re.match(p, t):
...         print("match")
...     if re.search(p, t):
...         print("search")
...
>>> t("is", target)
search
>>> t("This", target)
match
search
>>> t("Th", target)
match
search
>>> t("ing", target)
search
>>> t("^ing", target)
>>>
```

The last two examples show that a pattern for which search() is successful becomes unsuccessful if changed to require with **^** that the match occurs at the start of the string.

Any successful application of matching or searching returns a *match object*. This match object includes a number of useful methods, the most important of which are:

Method	**Description**	**Value Returned for p2 ("555-555-5555")**
group()	Returns the entire matched string.	555-555-5555
start()	Returns the start index of the match.	0
end()	Returns the end index of the match.	12
span()	Returns a tuple with the start and end indexes of the match.	(0, 12)

The match object returned from an **re.match()** call always has a start() value of 0 and the span() method also always returns 0 as the first element of the tuple. This is because, as noted earlier, the **match()** function only returns patterns found at the start of a string.

On the other hand, the **search()** function finds strings anywhere. The test_search() function in the tests calls check_number(p1), which calls **search()**. This also returns a match object, although it isn't returned to the caller. If we apply search() to paragraph 1, we see:

Method	**Description**	**Value Returned for p1**
group()	Returns the string matched.	"555-123-4567"
start()	Returns the start index of the match.	65
end()	Returns the end index of the match.	77
span()	Returns a tuple with the start and end indexes of the match.	(65, 77)

As you can see, **match()** and **search()** are two very similar functions with a single important difference.

More Regular Expression Features

The code you wrote found numbers of the form XXX-XXX-XXXX, because the **re** module's functions recognize "\d" as requiring a digit in the scanned string. (The backslash tells the functions that the "d" is to be specially interpreted—without it, they would only match the literal character "d"). But what about (555)-123-4567? That is a phone number, but it doesn't follow the same pattern.

You could write a second regular expression for this, and then try matching the first and only try the second if the first one did not match. This could become clumsy quite rapidly in the case of complex patterns. Fortunately, regular expressions can model complex patterns to handle this sort of problem. Regular expressions can specify using alternate patterns using the "|" special character, which means a pattern like the one below will find phone numbers following either the XXX-XXX-XXXX or (XXX)-XXX-XXXX patterns.

OBSERVE: Syntax for Regex With | (or) Matching

```
r"\d\d\d-\d\d\d-\d\d\d\d|\(\d\d\d\)(-| )\d\d\d-\d\d\d\d"
```

By now you are probably thinking that every character in a regular expression must be preceded by a backslash! This is not the case, but as we've learned, the parentheses have a specific meaning to the regular expression matching routines, so they need to be escaped to tell the routines to look for them just as regular characters.

One of the difficulties of the pattern above is that both alternate patterns have the same ending but different beginnings. We can overcome this by using parentheses to group portions of our pattern. So an equivalent pattern (ignoring complexities we haven't yet covered) would be

OBSERVE: Alternative Syntax for Regex With | (or) Matching

```
r"(\d\d\d|\(\d\d\d\))(-| )\d\d\d-\d\d\d\d"
```

In this pattern the alternation is restricted to the portions inside the **parentheses**—that is, the parentheses that are not preceded by backslashes. So the part of the pattern in parentheses will match either three digits or three digits surrounded by parentheses. Then it will match either a dash (-) or a space. In either case the rest of the pattern is the same. Now modify match_vs_search.py to use this extended pattern.

CODE TO EDIT: match_vs_search.py

```
"""
Demonstrate the difference between match() and search().
"""

import re

def check_number(text):
    ~~regex = r"\d\d\d-\d\d\d-\d\d\d\d"~~
    regex = r"(\d\d\d|\(\d\d\d\))(-| )\d\d\d-\d\d\d\d"
    match = re.match(regex, text)
    if match:
        return match.group()

    match = re.search(regex, text)
    if match:
        return len(text)
```

Save your changes. To correctly test this update, we also need to modify the test routine by adding tests that require correct matching of numbers whose area codes are in parentheses and followed by a dash or a space. You will see there is also some simplification of the test code, since there is no need to store the result in a variable before testing it.

CODE TO EDIT: test_match_vs_search.py

```
import unittest
from match_vs_search import check_number

p1 = """While I was at the store in Washington, DC 20001 I tried to call 555-123
-4567 on my mobile
but accidentally called 555-754-4321.  The person on the line redirected me to
999-999-9999 which I don't think is a real number. Neither is 000-000-0000 or 55
5-555-0000.
Well, I will try (555) 123-4567 again now."""

p1a = """While I was at the store in Washington, DC 20001 I tried to call (555)
123-4567 on my mobile
but accidentally called (555)-754-4321.  The person on the line redirected me to

(999)-999-9999 which I don't think is a real number. Neither is (000)-000-0000 o
r (555) 555-0000.
Well, I will try (555) 123-4567 again now."""

p2 = "555-555-5555"
p2a = "(555)-555-5555"

p3 = "What is the author's phone number?"

class TestRegex(unittest.TestCase):
    def test_match(self):
        ~~result = check_number(p2)~~
        ~~self.assertEqual("555-555-5555", result)~~
        self.assertEqual("555-555-5555", check_number(p2))
        self.assertEqual("(555)-555-5555", check_number(p2a))

    def test_search(self):
        ~~result = check_number(p1)~~
        ~~self.assertEqual(305, result)~~
        self.assertEqual(305, check_number(p1))
        self.assertEqual(315, check_number(p1a))

    def test_none(self):
        result = check_number(p3)
        self.assertIsNone(result)

if __name__ == "__main__":
    unittest.main()
```

Save and run it. If you have correctly modified your code, all tests should pass.

OBSERVE: Tests all pass.

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

More Complex Matching

So far, we've only seen a little of what regular expressions can do. It is quite easy to extend the searching facilities to alphanumeric patterns. Suppose we need to find a city, state, and zip code in a paragraph—the text would follow this rough pattern: *City Name, State Abbreviation Zip Code*. How do you express a pattern to match such strings?

The first thing we want to do is get the capital letter that starts each city name. In regular expressions, we can match a single occurrence from a set of characters by putting the characters in square brackets—to match any upper-case letter, we can use **[ABCDEFGHIJKLMNOPQRSTUVWXYZ]**. This is rather tedious to type, so we can use a range, **[A-Z]**, instead.

Next we need to match the other letters of the city name (we assume there will be one or more further characters). For that we'll use the brackets and range again, but add a little more: **[a-z]+**. The plus sign allows for any number of lower-case letters to match. So the pattern to match a capitalized word is **[A-Z][a-z]+**.

Some cities have multiple words in their name (Falls Church and San Francisco come to mind). Thus, the first word can optionally be followed by one or more further words, each separated from its predecessor by white space. So we need to follow the original pattern with zero or more repeats to the same pattern, with the repeats preceded by a whitespace. The pattern for that is **(\s[A-Z][a-z]+)***.

Note that, in order to apply the * character to the whole grouping, parentheses are required.

Now, we need to account for the state abbreviations. The easiest way to do it in regular expressions is via **[A-Z]{2}**, which only allows two uppercase letters, matching the US postal designation for American states. Add that to our regular expression, include a comma, and allow for a little white space: **,\s[A-Z]{2}**.

Finally, we handle zip code handling portion of the pattern. We won't check for nine-digit or foreign postal codes right now, so for our purposes, **\d{5}** will suffice. This makes the final pattern **[A-Z][a-z]+(\s[A-Z][a-z]+)*,\s[A-Z]{2}\s\d{5}**.

Now we'll try incorporating that into a function that finds the required addresses. Naturally, we need to write some tests first. In your **Python3_Lesson05/src** folder, create **test_city_search.py** as shown:

CODE TO TYPE: test_city_search.py

```
import unittest

from city_search import city_search

p1 = """While I was at the store I tried to call 555-123-4567 on my mobile
but accidentally called 555-754-4321.  The person on the line redirected me to
999-999-9999 which I don't think is a real number. Neither is 000-000-0000 or 55
5-555-0000.
Well, I will try (555) 123-4567 again now."""

p2 = "I live in Washington, DC 20002. Where do you live?"
p3 = "I live in Falls Church, VA 20188. And you?"

class TestRegex(unittest.TestCase):

    def test_city_search(self):
        self.assertEqual("Washington, DC 20002", city_search(p2))
        self.assertEqual("Falls Church, VA 20188", city_search(p3))

    def test_city_search_failure(self):
        self.assertIsNone(city_search(p1))

if __name__ == "__main__":
    unittest.main()
```

Save it. Most of the work has already been done with the design of the regular expression, and the function now simply needs to use it to locate addresses. Create **city_search.py** as shown:

CODE TO TYPE: city_search.py

```
"""
String regular expressions
"""

import re

def city_search(text):

    regex = r"[A-Z][a-z]+(\s[A-Z][a-z]+)*,\s[A-Z]{2}\s\d{5}"

    search = re.search(regex, text)
    if search:
        return search.group()
```

Save it, and then run **test_city_search.py**. The tests should pass.

OBSERVE: Results of Running test_city_search.py

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Regular expressions have a power which their apparent simplicity belies, as you can now start to appreciate.

Finding all with findall() and finditer()

The first programming example in this lesson used a regular expression function named **findall()**. In that code, it returned a list of non-overlapping matching phone numbers from the paragraph. This is useful for providing a list of strings, but what if you need to know the start and end index of each of those phone numbers, in other words familiar data shown below, but for each found part of the string? While findall returns a list of the matching strings, finditer returns a list of the matching objects, and each match object has these methods:

Method	Description
group()	Returns the string matched.
group(n)	Returns the string matched by the *n*th parenthesised group in the pattern.
group(m, n, ...)	Returns a tuple of the strings matched by the *m*th, *n*th, and so on parenthesized groups in the pattern.
start()	Returns the start index of the match in the target string (always 0 for **re.match()**).
end()	Returns the end index of the match.
span()	Returns a tuple with the start and end indexes of the match.

More on Modifying Strings With sub() and subn()

Suppose you don't want to publish all the phone numbers in this lesson, but you do want to show area codes. Regular expressions let you find patterns, and they also provide tools to allow you to modify them. The regular expression **sub()** method can make this sort of substitution. Pass in your pattern, what you want it replaced with, and the string to modify: **re.sub("\d\d\d-\d\d\d\d", "XXX-XXXX", text)**.

Let's make a program to show this in action. Create **test_phone_hide.py** in your **Python3_Lesson05/src** folder:

CODE TO TYPE: test_phone_hide.py

```
import unittest

from phone_hide import phone_hide

text = """While I was at the store I tried to call 555-123-4567 on my mobile
but accidentally called 555-754-4321.  The person on the line redirected me to
999-999-9999 which I don't think is a real number. Neither is 000-000-0000 or 55
5-555-0000.
Well, I will try (555)-123-4567 again now.
"""

class TestRegex(unittest.TestCase):

    def test_phone(self):
        response = phone_hide(text)
        self.assertFalse("555-123-4567" in response)
        self.assertTrue("555-XXX-XXXX" in response)
        self.assertTrue("(555)-XXX-XXXX" in response)

if __name__ == "__main__":
    unittest.main()
```

Then, create **phone_hide.py** in the same folder:

CODE TO TYPE: phone_hide.py

```
import re

def phone_hide(text):

    # Don't forget to use a raw string constant!
    return re.sub(r"\d{3}-\d{4}", "XXX-XXXX", text)
```

Save both programs and run the test:

OBSERVE: Running test_phone_hide.py

```
.
----------------------------------------------------------------------
Ran 1 test in 0.022s

OK
```

What if we want to know how many substitutions occurred? Then we can use the **subn()** function, which returns a two-element tuple containing the result string and the number of substitutions. Modify **test_phone_hide.py** as shown:

CODE TO EDIT: test_phone_hide.py

```
import unittest

from phone_hide import phone_hide

text = """While I was at the store I tried to call 555-123-4567 on my mobile
but accidentally called 555-754-4321.  The person on the line redirected me to
999-999-9999 which I don't think is a real number. Neither is 000-000-0000 or 55
5-555-0000.
Well, I will try (555)-123-4567 again now.
"""

class TestRegex(unittest.TestCase):

    def test_phone(self):

        response, count = phone_hide(text)
        self.assertFalse("555-123-4567" in response)
        self.assertTrue("555-XXX-XXXX" in response)
        self.assertTrue("(555)-XXX-XXXX" in response)
        self.assertEqual(6, count)

if __name__ == "__main__":
    unittest.main()
```

Save and run it:

OBSERVE: Running test_phone_hide.py

```
E
======================================================================
ERROR: test_phone (__main__.TestRegex)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\sholden\workspace\Python3_Lesson4\src\test_phone_hide2.py", lin
e 14, in test_phone
    response, count = phone_hide(text)
ValueError: too many values to unpack

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

It fails because the current **phone_hide()** function still returns a single value. We need to modify it to call **subn()** instead of **sub()**.

CODE TO EDIT: phone_hide.py

```
import re

def phone_hide(text):

    # Don't forget the 'r' at the start of the string!
    return re.subn(r"\d{3}-\d{4}", "XXX-XXXX", text)
```

Inserting that single letter "n" should be enough to restore everything to a fully functional state.

Save it, and run the test again:

OBSERVE: test_phone_hide.py now passes again

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Breaking Strings Apart with split()

Now suppose we want to split up a paragraph into a list of sentences. Python's **split()** function makes this problem trivial to solve. The regular expression pattern to find a sentence end (assuming some simplifications) is **r"[?.!]\s+"**.

The bracketed set contains the punctuation characters **?**, **.**, and **!**, which represents the ending of each sentence. Although these characters all have special meanings in regular expressions, remember that within a character set specification, they are treated as literal.

The '\s+' portion of the pattern requires a space or spaces after the ending punctuation of a sentence. The more precise you make a pattern the better your results will be. Without the spaces, a period used as a decimal point inside a number would be treated as ending a sentence.

Let's give it a try! As usual we'll begin by writing the tests. In your **Python3_Lesson05/src** folder, create **test_sentence_split.py** as shown:

CODE TO TYPE: test_sentence_split.py

```
import unittest

from sentence_split import sentence_split

text = "Hello! My name is Steve. What is yours? I hope you enjoyed this class!"

class TestRegex(unittest.TestCase):

    def test_split_sentence(self):
        numbers = sentence_split(text)
        self.assertEqual(len(numbers), 4)

if __name__ == "__main__":
    unittest.main()
```

Then, in the same folder, create **sentence_split.py**:

CODE TO TYPE: sentence_split.py

```
import re

def sentence_split(text):
    return re.split(r"[?.!]\s+", text)
```

Save both files, and run the test:

results of test_sentence_split.py

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Use Regular Expressions With Care

Regular expressions are extremely powerful. As you expand your knowledge, you'll be amazed by what they can do.

However, with great power comes great responsibility! So here are a couple of warnings about the use of regular expressions in programming.

Cn U Rd Ths?

Regular expressions can get extremely complex. For example, let's say you want to pull all of the email addresses from a paragraph. This sounds like a simple enough task, right? Something like **r"[a-zA-Z.-]+\@[a-zA-Z.-]+"** should work, right?

Unfortunately, if you apply that pattern to "**So... um...@oreilly we found his email was steve@oreilly.com.**" you will get **steve@oreilly.com** out, but it will also give you **um...@oreilly**!

So your pattern should be able to handle only proper email prefixes and should not allow repetition of dots. There are other specifications for allowed domain suffixes such as nations, .info, .com, and others. You should really research RFC 2822, which is the official email specification, with all its special cases and rules. And that sort of complexity generates regular expressions that look like this:

"[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+(?:[A-Z]{2}|com|org|net|edu|gov|mil|biz|info|mobi|name|aero|asia|jobs|museum)\b"

As you can see, regular expressions can get out of hand—and this is just for emails! Regular expression syntax is arguably not very clear compared to the elegance of Python, and it is not uncommon for authors of regular expressions to lose track of what their effort is supposed to do.

There are ways to make regular expressions more legible, but be aware of the code clarity issues that regular expressions can cause.

String Methods Versus Regular Expressions

"When the only tool you own is a hammer, every problem begins to resemble a nail."
-Abraham Maslow, American educator

You've just been introduced to the world of regular expressions, an amazingly powerful toolbox that can do incredible things. You've also been warned about the dangers of regular expressions. There is still much more to learn, and the Python documentation describes regular expressions in rather more detail (a confusing amount of detail for beginners, we suspect).

For over two courses and about thirty lessons, we've been able to rely on string methods. And that is because Python's string methods are fast and powerful, and yet easy to use. By all means, continue to use them when it is easy and faster to do so.

Sometimes regular expressions aren't the right tool for the job. Sometimes it pays to write a dozen lines of Python code instead of a single regular expression. There are no hard and fast rules to follow; it is just something that you learn over time.

For some reason, we find that regexes enthuse people to the point that they become the hammer with which they try to solve all string-processing problems. Don't let this happen to you.

In the next lesson, you'll learn that Python allows you to build regular expression pattern objects, which can make your code more compact and readable as well as more efficient.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Compiling and Flagging Regular Expressions

In preceding lessons, you learned about regular expressions and their basic use in the Python language. In this lesson, you'll learn about how to compile regular expression patterns, Python's special regular expression flags, additional pattern matching strings, and we'll try more examples. By the end of this lesson, you'll know enough to handle most of your regex needs.

This lesson includes these sections:

- Compiling Regular Expressions
- Flagging Regular Expressions

Compiling Regular Expressions

So far, we've used the module-level functions in Python's **re** library in order to do pattern matches. The advantage of this is that it makes for quick-to-write code, but from a performance point of view, it is not the most efficient method. For the small examples we've used so far, it hasn't been a problem, but regular expressions are often called in huge volumes on gigantic strings and the module-level functions have their limits. So when you anticipate a need for greater performance, it is a good practice to compile the regular expressions before use.

Compiled regular expressions are called a *pattern object*. All of your favorite Python re search functions are methods of the pattern object. Actually, many of these methods have additional features that the basic search functions lack, which allow you to really fine-tune your searches.

When you compile your patterns, since they are no longer strings, your code is more compact, more readable, and more usable.

Using re.compile() to Make a Pattern Object

To compile a regular expression into a pattern object, you pass a pattern string into the **re.compile()** function. Once you've done that, you can start using the **re** functions you've learned before, such as match(), search(), findall()—albeit now as methods:

CODE TO TYPE: Run the following code in an interactive terminal session

```
>>> import re
>>> regex = re.compile('Python')
>>> my_str = "I'm glad O'Reilly has Python courses and books!"
>>> result = regex.search(my_str)
>>> result
<_sre.SRE_Match object at 0x4b8e58>
>>> result.group()
'Python'
>>> regex.match(my_str) == None # Match fails because 'Python' is not at the sta
rt
True
>>> regex.findall(my_str)
['Python']
```

If you continue to play around with the pattern object, you'll see you can use **finditer()**, **sub()**, and **subn()** as methods. Indeed, the pattern object functionality matches that of the core **re** library functions.

Pattern objects and positional arguments

Actually, the statement '*the pattern object functionality matches that of the core* ***re*** *library functions*' is incorrect. The pattern object also includes for many of its methods *pos* and *endpos* arguments. These act just like string slicing, but if the endpos argument is less than the pos argument, the method returns a None object instead of an empty string on the match() and search() methods and an empty list/iterator for the findall() and finditer() methods, respectively.

CODE TO TYPE: Continuation of session above

```
>>> new_str = 'Python is a language; a Python is a snake'
>>> regex.findall(new_str)
['Python', 'Python']
>>> regex.findall(new_str, 6) # starts at position 6
['Python']
>>> regex.findall(new_str, 6, 10) # starts at position 6, ends at position 10
[]
>>> regex.findall(new_str, 10, 5)
[]
>>> type(regex.match(new_str, 10, 5))
<class 'NoneType'>
```

Not all pattern object methods include position arguments, so here is a reference guide:

Method	**Positional Arguments?**
search	yes
match	yes
split	no
findall	yes
finditer	yes
sub	no
subn	no

Flagging Regular Expressions

When you get into writing longer and more complex regular expressions, it becomes hard to read the pattern. Wouldn't it be nice to be able to be able to include comments in your regular expressions? Or spread the regular expression across multiple lines without creating false positives? Or ignore alphabet case by default? Or only Flags give you that and more.

Verbose Regular Expressions

Earlier, we used this regular expression to find cities in a text string:

[A-Z][a-z]+(\s[A-Z][a-z]+)*,\s[A-Z]{2}\s\d{5}

This is not very easy to read. Fortunately, we can break it up and still keep it usable, with the re.VERBOSE flag. Let's make an example. Create the **Python3_Lesson06** project and assign it to the **Python3_Lessons** working set. Then, copy **city_search.py** and **test_city_search.py** from the previous lesson into the **Python3_Lesson06/src** folder. Edit **city_search.py** as shown:

CODE TO EDIT: city_search.py

```
"""
String regular expressions
"""

import re

def city_search(text):

    ~~regex = r"[A-Z][a-z]+(\s[A-Z][a-z]+)*,\s[A-Z]{2}\s\d{5}"~~
    regex = re.compile(r"""
        [A-Z][a-z]+       # the first word of a city
        (\s[A-Z][a-z]+)*  # possible additional words of a city
        ,\s[A-Z]{2}\s     # The two-letter abbreviation for a US state
        \d{5}             # five-digit US zip code
        """, re.VERBOSE)

    ~~search = re.search(regex, text)~~
    search = regex.search(text)
    if search:
        return search.group()
```

Save it and run **test_city_search.py**. It still passes the tests.

As you can see, when the pattern object is compiled, you passed in re.VERBOSE as an extra argument. This argument allowed you to include white space and Python-style comments without breaking the regular expression.

The trick with this particular flag is that all the white space is removed, except that which is declared, so you need to remember to include \s, \n, \r, \f, \t, and \v instead of literal spaces, tabs, and return characters.

Ignoring Case

If you need to match a pattern and ignore case, the best way to do it is with the re.IGNORECASE flag:

CODE TO TYPE: Run the following code in an interactive terminal session

```
>>> import re
>>> regex = re.compile(r"""python # the language
... |guido # the bdfl
... """, re.IGNORECASE | re.VERBOSE)
>>> for m in regex.findall("""Python was invented by Guido, and while its mascot
 is a
... python, it was named after Monty Python"""):
...     print(m)
...
Python
Guido
python
Python
```

Note that when we pass in two flags, we use the pipe (|) symbol, thus: **re.IGNORECASE | re.VERBOSE**

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Python's Object-Oriented Features

Earlier in this course, you learned some of the basics of object-oriented programming (OOP), which was first discussed in "Beginning Python." In this lesson, you'll learn more about OOP, and understand more deeply the object-oriented features that Python offers.

Object-oriented programming is commonly held to be based on three fundamental concepts (also the sections in this lesson):

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

Encapsulation is the idea that the only way to access or change the data inside an object is by calling its methods. This idea has never really gained much ground in the Python world, and it is normally considered acceptable to both read and set an object's attributes from anywhere in a program.

Occasionally, you may find that storing new information in an object requires you to perform other calculations. While it might seem that a method call would be necessary in such circumstances, you can instead choose to perform the calculations by implementing a *property*, which we will show how to do later.

Inheritance

A quick subclassing review

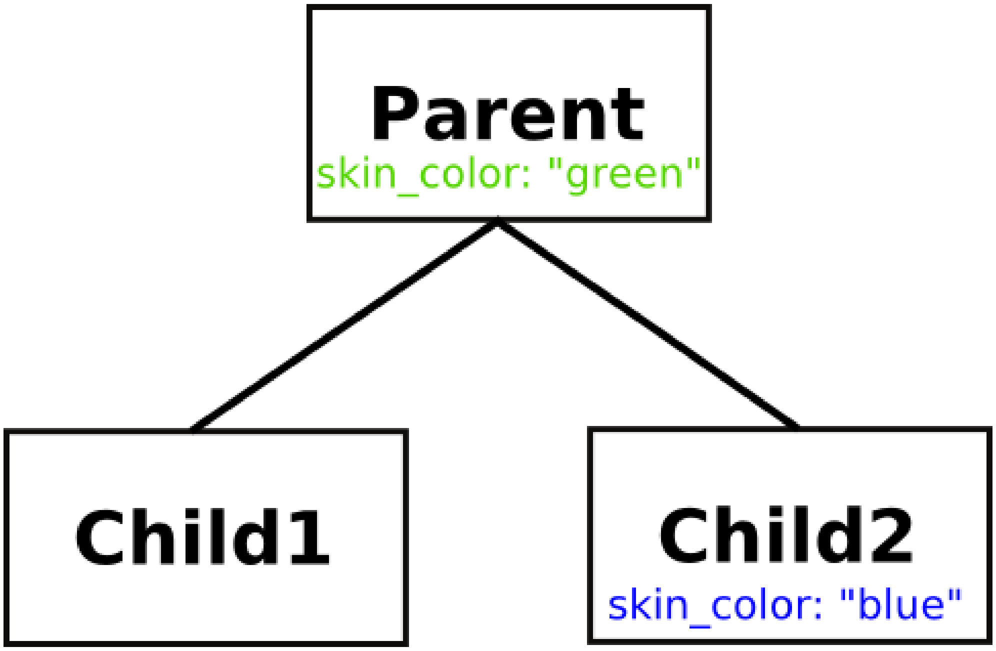
You have already used Python's inheritance features, so you know something about them. In programming, when a child class inherits from a parent class, that is referred to as *subclassing*. In Python, we say that the subclass (child) inherits from a base class (parent). To the programmer, it appears that the subclass has all of the same attributes (including methods) as the base class—though in fact this is actually implemented by the interpreter following a well-defined *method resolution order* (MRO) to locate attributes. Run an example in an interactive interpreter window as follows to clarify this.

CODE TO TYPE: Run the following code in an interactive terminal session

```
>>> class Parent:
...     skin_color = "green"
...
>>> class Child1(Parent):
...     pass
...
>>> class Child2(Parent):
...     skin_color =  "blue"
...
>>> Child1.skin_color
'green'
>>> Child2.skin_color
'blue'
>>> Child2.__mro__
(<class '__main__.Child2'>, <class '__main__.Parent'>, <class 'object'>)
>>> object
<class 'object'>
>>> dir(object)
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
>>> dir(Parent)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__g
e__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '
__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'skin_color']
>>> object.__dict__
<dict_proxy object at 0x024941B0>
>>> Parent.__dict__
<dict_proxy object at 0x024941B0>
>>> sorted(list(Parent.__dict__))
['__dict__', '__doc__', '__module__', '__weakref__', 'skin_color']
>>> sorted(list(Child1.__dict__))
['__doc__', '__module__']
>>> sorted(list(Child2.__dict__))
['__doc__', '__module__', 'skin_color']
>>> sorted(list(object.__dict__))
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
>>>
```

Note Some of the lines above have been wrapped for your reading convenience.

In the example above, the value of **Child1.skin_color** is "green," because if the interpreter doesn't find the attribute it is looking for in the class it will next look in its base class. The **Child2** class sets its own **skin_color**, however, and so when the interpreter looks for a "skin_color" attribute in the class's namespace, it finds it without any need to look in the parent class. We say that the **Child2** class *overrides* the base class's skin color.



The diagram above shows the inheritance relationship between Parent, Child1, and Child2, which makes it obvious why Child1 has green skin. You can see the MRO of a class by examining its __mro__ attribute, as is shown in the interactive session. This tuple is a list of the class's base classes. You will observe that although it is never explicitly mentioned in any of the class definitions, Python classes ultimately inherit from a built-in class called **object**, and that much of the behavior of your classes is actually defined in that class.

Note Technically, the built-in classes are usually referred to as *types*. There are a few differences between those types and the classes you define yourself, but you don't need to be concerned about them just yet.

You can also see that the names that have been defined locally to a class generally live in its **__dict__**. To a first approximation, the output of **dir()** on a class will be its **__dict__** plus the **__dict__**s of all its base classes. That is because the class's **__dict__** is where the class attributes are stored.

You might also notice that classes don't actually use a Python dict as their **__dict__**, but instead have a specialized object called a *dict_proxy*. This is a "lightweight" dict, designed to operate lookups as quickly as possible because name lookups are so frequent in Python.

One other term to remember: the base class that is the immediate parent of a class is often called its *superclass*.

Multiple Inheritance

Python implements *multiple inheritance*: you can specify more than one base class in a class definition, and your class will inherit the characteristics of all its base classes. This allows you to define classes called *mix-in classes* that you can use specifically to add behaviors to other classes.

This naturally gives rise to the question "what happens if more than one of the base classes defines the same attribute—which value does my class inherit?" As with so many questions about Python, the interactive interpreter is your friend. Let's use it to find out.

CODE TO TYPE: Investigate more complex inheritance in an interactive interpreter session

```
>>> class Mother:
...     hair_color = "blonde"
...     temperament = "placid"
...
>>> class Father:
...     hair_color = "ginger"
...     curiosity = "high"
...
>>> class Daughter(Mother, Father):
...     pass
...
>>> class Son(Father, Mother):
...     pass
...
>>> Daughter.hair_color
'blonde'
>>> Son.hair_color
'ginger'
>>> Daughter.temperament, Daughter.curiosity
('placid', 'high')
>>> Son.temperament, Son.curiosity
('placid', 'high')
>>>
```

The Daughter class inherits the Mother class's **hair_color** because the base classes are searched left-to-right. Similarly the Son class inherits the Father class's **hair_color**. However, both children inherit **temperament** from the Mother class and **curiosity** from the Father class, because only one base class defines each of these attributes. Inheritance of methods works in exactly the same way: in resolving a method or attribute name, the interpreter searches the base classes (and their subclasses, and so on) starting from the left—all subclasses of the first base class are considered before the second base class

.

Polymorphism

Polymorphism: Same Operations, Different Types

One of the concepts that Python supports very well is Subtype Polymorphism, known less formally as *polymorphism*. Polymorphism gives you the ability to write code without concerning yourself about the types of the data it is dealing with.

Early in this series of classes, you used Python to perform some basic math on integers, and eventually expanded your knowledge to understand that you could add (or more properly "concatenate") strings and various iterators together. Use the interactive interpreter to remind yourself again about this interesting property of Python.

CODE TO TYPE: Use the interactive interpreter to define a polymorphic function

```
>>> def add(x, y):
...     return x+y
...
>>> add(3, 5)
8
>>> add("big", "string")
'bigstring'
>>> add([1, 2, 4], [8, 16])
[1, 2, 4, 8, 16]
>>> add((1, 1, 1), (2, 2, 2))
(1, 1, 1, 2, 2, 2)
>>>
```

The above function demonstrates that in Python, numbers, strings, lists and tuples are *polymorphic with respect to addition*. As long as both arguments are of the same type, you can add them together.

Did you ever stop to wonder about how the + and * operators "know" how to do the correct operations on the operands on either side? If you think about it, the computer has to perform quite different operations to add two strings and two numbers. This polymorphism is achieved by examining and calling methods of the operands.

When the interpreter has to evaluate the expression **a + b**, it first tries to evaluate **a.__add__(b)**. This may or may not be possible: the **a** object may not *have* an **__add__()** method, or the method might return NotImplemented when called with **b** as an argument. In either of these cases, the interpreter falls back to trying to call **b.__radd__(a)** to evaluate the expression. If this is impossible (again, either because **b** has no **__radd__()** method, or because that method raises NotImplemented when called with **a** as its argument) the interpreter raises a TypeError exception.

One more thing before we explore actual usage—in an earlier lesson, we wrote code to determine a child's hair color. Our tests checked the response of an expected **hair()** method. This was yet another example of polymorphism.

Let's create a working example that does use polymorphism. You've got a farm and you need to list all the animals, the sounds they make, and whether they have wings. Create the Pydev project for **Python3_Lesson07** and assign it to the **Python3_Lessons** working set. Then, in the **Python3_Lesson07/src** folder, create **test_animal_farm.py** as shown:

CODE TO TYPE: test_animal_farm.py

```
'''
Test the animal_farm animals
'''
import unittest
from animal_farm import Animal, Pig, Dog, Chicken

class Test(unittest.TestCase):

    def test_base_animal_class(self):
        "Tests the basics of the Animal class."
        animal = Animal("Orwell")
        self.assertRaises(NotImplementedError, animal.sound)
        self.assertFalse(animal.has_wings())

    def test_pig(self):
        "Tests the inhabitants of the farm"
        pig = Pig("Napoleon")
        self.assertEqual(pig.sound(), "oink!")
        self.assertFalse(pig.has_wings())

    def test_dog(self):
        dog = Dog("Bluebell")
        self.assertEqual(dog.sound(), "woof!")
        self.assertFalse(dog.has_wings())

    def test_chicken(self):
        chicken = Chicken("Kulak")
        self.assertEqual(chicken.sound(), "bok bok!")
        self.assertTrue(chicken.has_wings())

if __name__ == "__main__":
    unittest.main()
```

The tests first determine that the base animal class works as expected. Then the individual animal classes are tested to make sure that they return the right sound and the right answer to the wing question.

Note that the **Animal** class's **sound()** method raises a NotImplementedError. This is a reminder that we assume all farm animals make sound, and the developer writing classes representing the beasts needs to implement this method. In programming parlance, the sound() method is called an *abstract method*. It doesn't do anything besides inform developers looking to use the **Animal** class what they need to do to make the class function correctly, and requires subclasses to implement the method.

The **has_wings()** method is different, assuming that most of the farm animals will by default not have wings, and so provides a default return of "False."

Now we need to create some animal classes to match the tests. In the **Python3_Lesson07/src** folder, create **animal_farm.py** as shown:

code to enter: animal_farm.py

```
class Animal(object):

    def __init__(self, name):
        self.name = name

    def sound(self):
        raise NotImplementedError("Animals need a sound method")

    def has_wings(self):
        return False

class Pig(Animal):

    def sound(self):
        return "oink!"

class Dog(Animal):

    def sound(self):
        return "woof!"

class Chicken(Animal):

    def sound(self):
        return "bok bok!"

    def has_wings(self):
        return True
```

Save the files and run the tests, and you have a working example of polymorphism.

OBSERVE: All four tests should pass

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

While tests are good to have, it's nice to see the actual application working too! Let's try this out on the command line:

OBSERVE: animal_farm in action

```
>>> from animal_farm import *
>>> animal = Animal('Mystery Meat')
>>> animal.name
'Mystery Meat'
>>> animal.sound()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "animal_farm.py", line 7, in sound
raise NotImplementedError("Animals need a sound method")
NotImplementedError: Animals need a sound method
>>> dog = Dog('Rover')
>>> dog.name
'Rover'
>>> dog.sound()
'woof!'
```

Overriding vs. Extending methods

You saw earlier how the definition of an attribute in a class will be chosen in preference to the definition of the same attribute in a base class. This is due to the method resolution order adopted in Python. When searching for an attribute (including a method), the interpreter first looks in the instance's namespace; next it looks in the namespace of the instance's class; after that it looks in the base classes one by one, raising an AttributeError exception if the attribute is not found.

If a class defines a method of the same name as a method of one of its base classes, it is said to *override* the method of the base class. So in the example above, the **Chicken** class's **has_wings()** method overrides the **Animal** class's **has_wings()** method, by providing its own implementation.

Sometimes, however, the subclass needs to use its superclass's method as a part of implementing its own method, and Python has a special feature to easily let you refer to a class's superclass—the **super()** function. You will see it in use in the next example, where we start by defining a **Car** class and then extend it by subclassing. The **Toyota** subclass needs an extra argument to its **__init__()** method, but it also needs to go through the usual initialization for cars. Create **test_extend.py** in your **Python3_Lesson07/src** folder as shown

CODE TO TYPE: test_extend.py

```
'''
test_extend.py: verify that Ford successfully
                extends the Car. __init__() method
'''
import unittest
from extend import Car, Ford, Toyota

class TestCars(unittest.TestCase):
    def test_Toyota(self):
        car1 = Car("red", 2000)
        car2 = Toyota("red", 2000, "Corolla")
        self.assertEqual(car1.color, car2.color)
        self.assertEqual(car1.cc, car2.cc)
        self.assertEqual(car2.model, "Corolla")

    def test_Ford(self):
        car1 = Car("red", 2000)
        car2 = Ford("red", 2000, "Taurus")
        self.assertEqual(car1.color, car2.color)
        self.assertEqual(car1.cc, car2.cc)
        self.assertEqual(car2.model, "Taurus")

if __name__ == '__main__':
    unittest.main()
```

The idea is that Toyotas are cars and Fords are cars, so they should use the **Car.__init__()** method to do the initialization that they have in common to set the instance variables. Observe that both the **Toyota** and **Ford** classes take an extra argument when you create a new instance, so clearly **Car.__init__()** alone is not going to suffice. The two subclasses are quite similar, differing only in the way they call their superclass's **__init__()** method. Now, create the **extend.py** program as shown:

CODE TO TYPE: extend.py

```
'''
extend.py: demonstrate how to extend a superclass method.
'''

class Car:
    def __init__(self, color, cc):
        self.color = color
        self.cc = cc

class Toyota(Car):
    def __init__(self, color, cc, model):
        Car.__init__(self, color, cc)
        self.model = model

class Ford(Car):
    def __init__(self, color, cc, model):
        super().__init__(color, cc)
        self.model = model
```

Note that the **Toyota** class's **__init__()** method calls **Car.__init__()** directly. Since **Car** is a class and not an instance, it is necessary to provide an explicit instance to the call.

The **Ford.__init__()** method, however, uses the built-in **super()** function. This returns a special object that delegates the calls to the parent class without needing an instance to be provided. If the tests all pass, that demonstrates that the two classes are equivalent in operation.

Note In version 2.7 of Python, super() has a different syntax (with arguments).

OBSERVE: Results of running test_extend.py

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

So both the subclasses, in their own way, extend the **Car.__init__()** method.

After this more extended look at Python's object-oriented features, you are better prepared to deploy the language to solve real-world problems. In the next lesson, we'll take a look at the features the language has for reading and storing data in compact binary formats.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Consuming and Creating Binary Data

This lesson includes these sections:

- Python Data vs. Raw Computer Data
- The struct Module

Python Data vs. Raw Computer Data

You have learned about many different data types that Python can deal with. This lesson explains how Python can be persuaded to exchange data with arbitrary programs, over whose data you have no control.

So far, all the external data (coming to and from files or the console) has been *character* or *string* data. You have handled this data without really needing to understand how it is represented, and now it is time to think about that a little. This requires you to understand something about character data to start with.

For quite a long time, the computer industry got by using character sets with only a limited number of characters. This was acceptable because people in most countries were writing programs for local consumption, and so they could encode their local alphabet so that each character was mapped onto one of the possible values of a byte. (A byte contains eight bits, so there are 256 different possible values from 0 to 255). Such a mapping is often referred to as an *encoding* of a character set. There has to be an agreement that different programs will treat the same byte values as representing the same character. For a long time, Python's string type used the *US ASCII* character set, using one byte to represent each character.

Then, the realization dawned that computer programs would eventually need to be capable of handling multiple languages, and in the mid-1980s work began on a way to encode much larger character sets, with the ultimate intention of being able to represent any text at all. This work ultimately led to the development of a standard called *Unicode*, which is what Python now uses to represent its strings. Internally, this requires the interpreter to represent each character as one to four bytes, using an encoding known as UTF-8. Python also provides support for many other encodings.

Unicode is not the most memory-efficient way to represent strings, and so for external storage and transmission a number of different ways of representing Unicode strings (normally referred to as *encodings*, just like ASCII) have been devised. Probably the most common in the Western world is UTF-8, which has been specially devised so that Unicode strings containing only ASCII characters will encode into the equivalent ASCII strings. The Python installed in Ellipse makes a default assumption that the external encoding of the text strings that it reads is UTF-8, but you may find that other Python interpreters have been configured to expect other encodings. You can ask the interpreter how it has been configured by calling **sys.getdefaultencoding()**, and you can determine the assumption it makes about the contents of text files by calling **sys.getfilesystemencoding()**. The two will not necessarily be the same, as you can see:

CODE TO TYPE: Type this code in an interactive Python console session

```
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
>>> sys.getfilesystemencoding()
'mbcs'
```

There are, however, times when it's important to be able to communicate in other than character terms. Sometimes, for example, you will receive a binary file and a description of its layout, and you will need to convert that data into the necessary Python types in order to be able to operate on it. Sometimes you will need to write your Python data out in a format required by other programs, with "raw" computer data types rather than string-based representations.

How Computers Represent Data

Most computers only work with a very limited set of different types of data: integers (of various sizes), floating-point numbers (of various sizes), and (sometimes) strings of bytes. Data types like Python's dicts and lists are not dealt with directly by the central processing unit (CPU). That is what the interpreter is for: it is a special-purpose program specifically designed to give you the impression that Python's data types are built in.

If you were to look at the layout in memory of a Python floating-point number, for example, you would see that it is far more complicated than a regular floating-point number used by the CPU. This is because the interpreter must maintain a bunch of overhead to do things like keeping track of how many references there are to an object (so the memory it uses can be reclaimed when it is no longer in use). But programs in other languages would not be able to make any sense out of Python's representation; they just want the data without any of that overhead.

So this leads to the interesting question of how the CPU actually represents the basic data types it *is* capable of dealing with. Fortunately "there's a module for that" in Python: the **struct** module (discussed later). It allows you to build memory structures (Python *bytes* objects) that can be written out to files or transmitted across networks for consumption by other programs.

The byte is the smallest addressable unit of memory in a modern computer and, as mentioned above, holds eight bits. A bytes object is a sequence of bytes, and so it can be subscripted and sliced just like strings and lists. When you open a file in binary mode and read data from it, what you get back is a bytes object. No decoding takes place on input, and no encoding on output. When a bytes object is read or written, you get the data transmitted, with no attempts to change it.

Note Python also implements a *bytearray* type. This is similar to the bytes type, but unlike strings and bytes, the bytearray is mutable, so you can change individual bytes by indexing, or sub-arrays by slicing.

The bytes and bytearray objects allow you to map the individual bytes of a file's contents, or of a sequence of bytes read over the network. The **struct** module allows you to interpret these values as the computer's basic data types—bytes, integers, and floating-point numbers.

The memory that your program works with (under the hood, that is, rather than the Python data types) is like a large bytearray, and the index of each byte is usually called its *address*. Addresses start, like Python indexes, at zero and go up by ones.

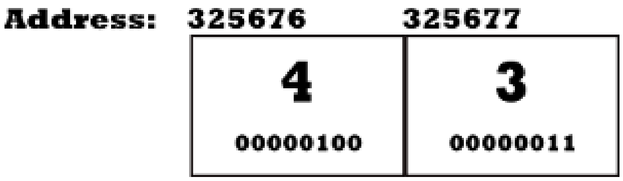
Endianness

The numbers that computers can deal with have grown bigger over the years. The more bits a number has, the larger the range of values it can represent. In modern computers, integers (whole numbers) will typically be represented as four bytes (though with the emergence of 64-bit computers, they can also be eight bytes). In older machines, they would be two bytes, now often referred to as a "short." Furthermore, integers can be either signed or unsigned, the former being able to represent both positive and negative values, the latter always interpreted as positive values.

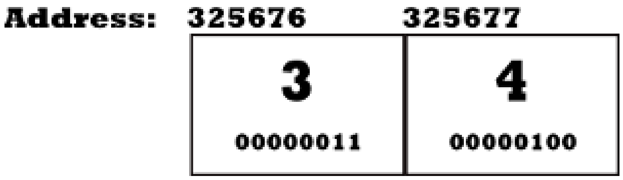
There are two principal ways to store numbers, known (for reasons we need not go into) as "big-endian" and "little-endian". The difference between them is the way that the bytes are stored: in a big-endian system, the most significant byte of a number is stored at the lowest memory address; in a little-endian system, it is stored at the highest memory address. For simplicity, let's consider a 16-bit (2-byte) representation of the number 1027.

The most significant byte will have the value 4, and the least significant byte will have the value 3 because **1027 = (4 * 256) + 3**.

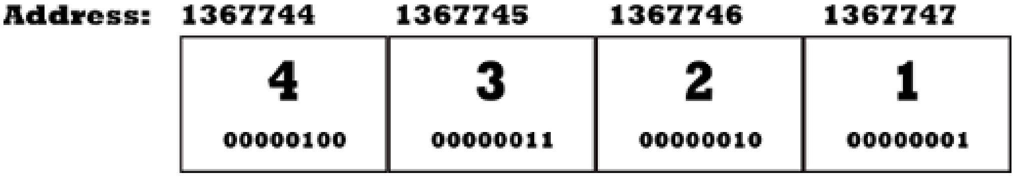
If this is stored at address 325676 in your program's memory, on a big-endian system it would look like this:



On a little-endian system, the same value stored at the same address would look like this:



This might not seem like much of a difference, but you have to know which endianness the data has when you are dealing with numbers made up of more than one byte. Otherwise you will interpret the numbers wrongly. The same thing occurs with longer values, though the arithmetic involved is more complex. Suppose you had the following bytes stored in memory starting at address 1367744.



Address:	1367744	1367745	1367746	1367747
	4 00000100	3 00000011	2 00000010	1 00000001

If this were a big-endian number, its most significant byte would be the 4 shown on the left, and its value would be **(((4*256+3)*256+2)*256+1 = 67,305,985**.

If it were little-endian, however, its most significant byte would be the 1 on the right, making its value **(((1*256+2)*256+3)*256+4 = 16,909,060**.

This should, we hope, convince you of the necessity to understand which type of data you are dealing with, since to deal with it the wrong way will lead to values that are just plain wrong!

Data Alignment

Yet another factor to take into account is the alignment of data. It is common for data to be aligned so that their starting address is a whole multiple of their size, so long (4-byte) integers will always be stored at an address that is an even multiple of 4, and so on.

These alignment rules are usually advisory rather than mandatory, but they are important: due to the way memory access works, it can take several times as long for the computer to add two non-aligned integers as it does to add two correctly-aligned ones. It's important to note that if the data are aligned this way, there may be so-called "packing" bytes inserted between values of different sizes. If you fail to take account of this, you will end up using the wrong bytes!

The struct Module

The **struct** module has been designed specifically to allow you to handle chunks of data that have been stored or transmitted in binary form to your Python program. Typically, you will read the data either from a file opened in binary mode or across a network connection. The module provides an **unpack()** function to let you interpret binary data and convert it to the appropriate Python data types. Its **pack()** function does the opposite, taking various Python data and converting them to a bytes object that can be stored or transmitted for other programs to interpret.

Format Strings

Both **pack()** and **unpack()** require a description of the data types in the bytes. This is presented as what the documentation refers to as a format string, whose first character is used to indicate the endianness of the data. In the following table, "native" means according to the rules of the particular computer on which the program is running. "Standard" alignment simply uses no packing bytes no matter whether items are correctly aligned or not. If the first character is none of those shown, it is assumed to be part of the format, and native settings are assumed.

First Character	Endianness	Packing
@	Native	Native
=	Native	Standard
<	Little-endian	Standard
>	Big-endian	Standard
!	Network (same as big-endian)	Standard

The remainder of the format string is a description of the individual data items that appear in the bytes object (for unpacking) or that are to be placed into the bytes object (for packing). The format characters can be preceded by a number, which indicates the number of values of that type to expect (except when the format character is "s," in which case it indicates the number of bytes in the string. This table shows the meanings of the various format characters.

Format	C Data Type	Python Type
x	Pad byte	-
c	char	bytes (length 1)

b	signed char	integer
B	unsigned char	integer
?	_Bool	bool
h	short	integer
H	unsigned short	integer
i	int	integer
I	unsigned int	integer
l	long	integer
L	unsigned long	integer
q	long long	integer
Q	unsigned long long	integer
f	float	float
d	double	float
s	char[]	bytes
p	char[]	bytes
P	void*	integer

If you aren't a C programmer, the "C types" may not mean that much. All you really need to know is that the unsigned types will always give positive values, and that if you try to pack a value that's too large to be held in the field, the interpreter will raise an exception.

Packing and Unpacking Values

One advantage of passing values in their binary form rather than as characters is that the representation will be exact, as bit-for-bit copies always are. Here is a demonstration that storing floating-point data in character form and reading it back can introduce small inaccuracies. There is no test code for this, since it is a simple demonstration program (it could be cast as a test, but this might obscure the actual differences in values). Create the **Python3_Lesson08** project and assign it to your **Python3_Lessons** working set. Then, in the **Python3_Lesson08/src** folder, create **floattest.py** as shown:

CODE TO TYPE: floattest.py

```
"""
floattest.py: checks for inaccuracies in floating-point test representations.
"""
import random, os
rlist = [random.random() for i in range(10)]
filename = r"V:\floatdata.txt"
f = open(filename, "w")
for x in rlist:
    print(x, file=f)
f.close()
f = open(filename)
for i in range(10):
    x = float(f.readline())
    if x != rlist[i]:
        print(i, x, rlist[i], abs(x-rlist[i]))
    else:
        print(i, x, "values agree")
print(filename, os.stat(filename).st_size)
f.close()
```

Save and run it. The program uses random numbers, and is therefore not entirely reproducible. A typical run, however, looks like this:

OBSERVE: Text representations of floating point may be less accurate than you think

```
0 0.308013405042 0.308013405042 7.58837437331e-14
1 0.383104050277 0.383104050277 2.5923707625e-13
2 0.279337151492 0.279337151492 3.80695475144e-13
3 0.262911769705 0.262911769705 4.72399896978e-14
4 0.97192333336 0.97192333336 4.15112388907e-13
5 0.535110192091 0.535110192091 2.10942374679e-13
6 0.453739263223 0.453739263223 4.61797267093e-13
7 0.346532896806 0.346532896806 2.92266211233e-13
8 0.237582673656 0.237582673656 3.80195874783e-13
9 0.157670914981 0.157670914981 8.15458811587e-14
V:\floatdata.txt 160
```

If you think about it, this seems completely weird: the program is telling you that (to take the first line as an example) 0.308013405042 differs from 0.308013405042 by 7.58837437331e-14. Now, that is a very small difference—another way to write it is 0.0000000000000758837437331, which is probably an error in the very last bit of the number. But any avoidable error is bad. It is obvious that there are two very slightly different numbers that Python represents as the string "0.308013405042." Could we avoid those errors by using the **struct** module? Edit the program as shown:

CODE TO EDIT: floattest.py

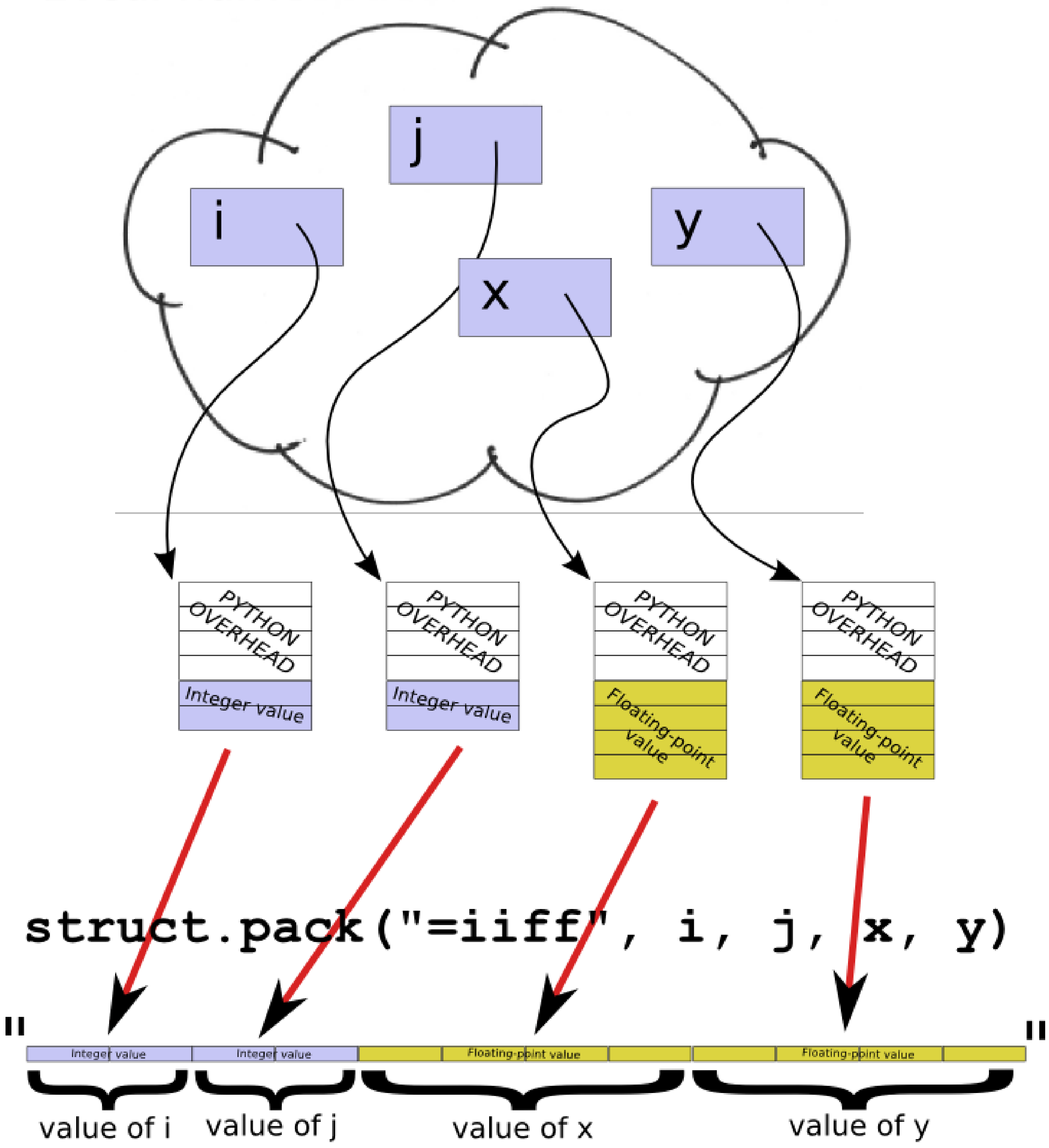
```
"""
floattest.py: check for inaccuracies in floating-point test representations.
"""
import random, os, struct
filename = r"V:\floatdata.bin"
rlist = [random.random() for i in range(10)]
f = open(filename, "wb")
f.write(struct.pack("=10d", *rlist))
~~for x in rlist:~~
~~    print(x, file=f)~~
f.close()
f = open(filename, "rb")
for i in range(10):
    s = f.read(8)
    ~~x = float(f.readline())~~
    x, = struct.unpack("=d", s)
    if x != rlist[i]:
        print(i, x, rlist[i], abs(x-rlist[i]))
    else:
        print(i, x, "values agree")
print(filename, os.stat(filename).st_size)
f.close()
```

Save and run it. The code uses the **struct.pack()** function to convert ten floating-point numbers (the elements of **rlist**, represented as positional arguments by the use of the * argument syntax) to fixed-length byte strings, which are written out to the (binary) floatdata.bin file. Next, it reads back eight bytes at a time, converting each bytes object back into a Python float. The output of this program is much more reassuring.

OBSERVE: Binary representation appear to be exact

```
0 0.505352274992 values agree
1 0.560349256654 values agree
2 0.86326435433 values agree
3 0.775838375892 values agree
4 0.498425623965 values agree
5 0.577260996053 values agree
6 0.247810402776 values agree
7 0.473451623047 values agree
8 0.184083222943 values agree
9 0.388145971055 values agree
V:\floatdata.bin 80
```

A further interesting fact is that the (slightly inaccurate) text file is roughly twice as large as the completely accurate binary file (remembering the random nature of the data, your result for the text file may be slightly different).



What struct.pack actually does: Pure CPU data are extracted from Python values and saved in their simplest form as part of struct.pack's result bytestring. With today's four-byte integers and eight-byte floating-point numbers the result of this call, with two integers and two floats, will be 24 bytes long.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Advanced Objects: Special Methods

Since you started writing classes and creating objects in Python, you've become familiar with the **__init__()** method in the initialization of the object to set up the object data. You've also learned about the **__add__()** method. Methods whose names start and end with double underscores ("__") designate special behaviors for Python classes via what are called *special methods*.

These special method names are tied directly into Python's infrastructure. They control how objects are created and destroyed, how they render through the **print()** function, and many other things. Their advantage is that they let you do a lot of very interesting, almost magical things with Python classes—which is why an alternative name for special methods is *magic methods*. Fortunately, the magic is like stage magic. Wonderful things seem to "just happen," but behind the scenes very explicit things are taking place, based on the way the interpreter has been designed. The wonderful part is that you can define your own objects to interact with the interpreter in pretty much the same way that Python's built-in objects do.

This lesson includes the following sections:

- Basic Customization
- Attribute Access
- Emulating Functions: the __call__() Method

Basic Customization

The most commonly used special methods are **__init__()**, **__new__()**, **__repr__()**, and **__str__()**. You've already used the **__init__()** method many times to initialize instance variables when new objects are created, so now we'll focus on the others. For each method. we'll provide some descriptive information, and then will include a brief example at the end to help you familiarize yourself with it.

__new__(): Creating New Objects

At first glance, The **__new__()** method seems similar to the **__init__()** method, but it is actually quite different. You will remember that you *instantiate* a class (that is, create a new instance of that class) by calling the class. The **__init__()** method returns nothing—it merely initializes what **__new__()** has created. The **__new__()** method, on the other hand, returns the object that will become the return value of the instantiation call.

Like **__init__()**, **__new__()** receives the arguments that the caller passes when calling the class. Unlike **__init__()**, **__new__()** receives a first argument that is the *class to be created* rather than the newly-created instance.

This is important: the default **__new__()** method (inherited from the *object* type) can be used to create immutable object instances. Most of the time you use this when extending immutable built-in types like numbers and strings, since it would not be possible to change them in the **__init__()** method. In our example, we'll create **ustr**, an extension of the basic **str** type that returns a string object that always has upper-case versions of any letters it may contain. Create the **Python3_Lesson09** project and assign it to the **Python3_Lessons** working set. Then, in the **Python3_Lesson09/src** folder, create **newmagic.py** as shown:

CODE TO TYPE: newmagic.py

```
"""
Python classes with magic methods
"""

class ustr(str):
    "An upper case string object."
    def __new__(cls, arg):
        arg = str(arg)
        return str.__new__(cls, arg.upper())
```

Before we continue, let's look closely at this class.

OBSERVE: The ustr Class

```
class ustr(str):
    "An upper case string object."
    def __new__(cls, arg):
        arg = str(arg)
        return str.__new__(cls, arg.upper())
```

This example defines the class **ustr** as a subclass of the built-in **str** type used to represent Unicode strings. Because Python's type names are lower-case, we break from the tradition of naming a class in MixedCase, and use a lower-case class name. The class defines a **__new__()** method that accepts **cls** and **arg** arguments.

The **cls** parameter is the actual class that was called—this is different from the *self* argument passed to other methods, which represents the instantiated object. Like *self*, the **cls** argument is provided automatically by the interpreter. The whole purpose of the **__new__()** method is to *create and return* the new object: this method is directly responsible for instantiation!

The **arg** parameter is the argument provided to the class when it is called. The value of **arg** is converted into a string, and the last statement returns a new string. It does so, however, by calling the built-in string type's **__new__()** method explicitly, asking it (with the first argument) to return an object of the correct type. The second argument to **str.__new__()** provides the value for the string, and upper case is guaranteed by calling its **.upper()** method.

The call to **str.__new__()** is analogous to an explicit call on a class method giving an instance as the first argument. The call to **str.__new__()** returns a **ustr** object, because the first argument to **__new__()** specifies the return type required. Test the class at the interactive console:

CODE TO TYPE: Test your ustr class at the interactive prompt

```
>>> from newmagic import *
>>> s = ustr("Steve Holden")
>>> s
'STEVE HOLDEN'
>>> type(s)
<class 'newmagic.ustr'>
>>> s.lower()
'steve holden'
>>> len(s)
12
>>> s.size = 12
>>> s.size
12
>>> ss = "A regular string"
>>> ss.size = 16
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'size'
>>>
```

You can see that instances of the **ustr** class behave almost the same as instances of **str**, except that they have to be created with an explicit call to the class. There is also the difference that you can set new attributes on **ustr** instances, but you cannot do that with the built-in type.

Representing objects as strings: __str__()

The built-in **str()** seems to be a function, but strictly speaking, it is actually a built-in type that can be *used* like a function. When you call it with a single argument, it tries to return a string representation of the argument by calling the argument's **__str__()** method. The **print()** built-in function does the same thing.

The return value of **__str__()** must be a string object. In our example below, the **Person** class is a normal class and the **NamedPerson** class provides a more attractive print statement. Create **strmagic.py** as shown:

CODE TO TYPE: strmagic.py

```
"""
Demonstrate string representations using inheritance
"""
class Person:
    "Represents a person"
    def __init__(self, name):
        self.name = name

class NamedPerson(Person):
    "Represents a person using their name"
    def __str__(self):
        return self.name
```

The difference between the two classes is that **NamedPerson** has an **__str__()** method, which **Person** does not. You can see the difference quite easily in an interactive interpreter session.

CODE TO TYPE: Test your objects in the interactive interpreter

```
>>> from strmagic import *
>>> p1 = Person("Danny Greenfeld")
>>> p1
<strmagic.Person object at 0x01E1D710>
>>> print(p1)
<strmagic.Person object at 0x01E1D710>
>>> p2 = NamedPerson("Danny Greenfeld")
>>> p2
<strmagic.NamedPerson object at 0x01E1D850>
>>> print(p2)
Danny Greenfeld
>>>
```

The string returned by **__str__()** is supposed to be an "informal" representation of the object, which can be used to convey its principal characteristics without necessarily allowing you to reproduce the object exactly. For the latter purpose, Python expects your objects to provide another magic method, **__repr__()**.

__repr__()

The **__repr__()** method of object *o* is called by **repr(o)**. The built-in function **repr()** is supposed to represent the "official" string representation of an object. Ideally, this representation should look like a valid Python expression which, when evaluated, produces the object being represented. Its primary use is in debugging or logging, and is best not revealed to users. The information should be as rich as possible.

The **str()** representation of a container object such as a list or a tuple also represents the contained objects using their **repr()** representation. Containers themselves generally use the same representation for both **str()** and **repr()**, and this is the easiest way to ensure that their representations are meaningful.

To determine a little more about the relationship between the two representational methods, we'll create four different classes that have different combinations of those methods. You can then see how they interact with the interactive interpreter and the **print()** function. (Don't forget, if you have other questions about this, the interactive interpreter is the best way to answer those questions). Create **reprmagic.py** as shown:

CODE TO TYPE: reprmagic.py

```
"""
Demonstrate differences between __str__() and __repr__().
"""

class neither:
    pass

class stronly:
    def __str__(self):
        return "STR"

class repronly:
    def __repr__(self):
        return "REPR"

class both(stronly, repronly):
    pass

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name
    def __repr__(self):
        return "Person({0.name!r}, {0.age!r})".format(self)
```

The **neither** class simply inherits all its behavior from Python's fundamental object. The **stronly** class, as its name implies, only implements **__str__()**, while the **repronly** class only implements **__repr__()**. The **both** class uses multiple inheritance to implement both methods. Finally, the **Person** class represents its instances as a string by using the instance's name and provides a full representation that could actually be pasted into a Python program as code. Note that it uses the **!r** format effector to include the formal representations of the instance's name and age. This avoids any tricky problems of representing strings with characters inside them that require escapes and so on.

CODE TO TYPE: Enter the following code in an interactive interpreter session

```
>>> from reprmagic import *
>>> o1 = neither()
>>> print(str(o1), repr(o1))
<reprmagic.neither object at 0x01E1DAD0> <reprmagic.neither object at 0x01E1DAD0
>
>>> o2 = stronly()
>>> print(str(o2), repr(o2))
STR <reprmagic.stronly object at 0x01E1DBB0>
>>> o3 = repronly()
>>> print(str(o3), repr(o3))
REPR REPR
>>> o4 = both()
>>> print(str(o4), repr(o4))
STR REPR
>>> o1
<reprmagic.neither object at 0x01E1DAD0>
>>> o2
<reprmagic.stronly object at 0x01E1DBB0>
>>> o3
REPR
>>> o4
REPR
>>> steve = Person("Steve Holden", 21)
>>> print(str(steve), repr(steve))
Steve Holden Person('Steve Holden', 21)
>>> tim = Person('Tim O\'Reilly', 55)
>>> tim
Person("Tim O'Reilly", 55)
>>>
```

In the lines where we asked the interactive interpreter directly for the objects o1 through o4, it presented the **repr()** of the objects. Remember that this behavior is specific to the interpreter's interactive mode: if you write an expression on its own in a Python module that is run as a main program, the interpreter simply calculates the value of the expression. Also note, from the example of the **repronly()** object bound to **o3**, that if an object has a **__repr__()** method but no **__str__()** method, the **__repr__()** method is used for both purposes.

Attribute Access

Attributes are where objects store data. Python lets you override the interpreter's normal attribute-handling behaviors by providing further special methods: **__getattr__()**, **__setattr__()**, and **__delattr__()** are used to access, set, and delete attributes respectively. These methods should be defined with great care: it is quite possible to end up with completely unusable objects if you are not sufficiently careful, or (even worse) objects that seem to do what you want them to but under certain circumstances don't behave as planned. As with **__str__()** and **__repr__()**, there are functions that you can use to access an object's special methods for attribute access, summarized here:

Function Call	Description
hasattr(o, name)	Returns **True** if object o has an attribute whose name is the same as the **name** argument, (which must be a string), otherwise returns **False**.
setattr(o, name, value)	Sets object **o**'s **name** attribute to **value** (provided that the object allows it). Equivalent to **o.__setattr__(name, value)**.
delattr(o, name)	If object **o** has an attribute called **name**, deletes the attribute. If no such attribute exists, raises an AttributeError exception. Equivalent to **o.__delattr__(name)**.
gettatr(o, name[, default])	If object **o** has an attribute whose name is the same as the call's **name** argument (which should be a string), returns its value. If no such attribute exists, returns the default (if it is provided in the call); if no default is provided, raises an AttributeError exception.

__setattr__()

Normally, when you set an attribute on an object, the name is used as a key and the value is stored in the object's __dict__, a special attribute used specifically to store instance variables. This method is called each and every time an attribute is set. In the code sample below, we use this feature to print a message each time an attribute is set. Create **attrmagic.py** in your **Python3_Lesson09/src** folder as shown:

CODE TO TYPE: attrmagic.py

```
"""
Demonstrate magic methods for attribute access.
"""
class AttrMixin:
    "Displays a message when an instance's attributes are set."
    def __setattr__(self, key, value):
        print("ATTR: setting attribute {0!r} to {1!r}".format(key, value))
        self.__dict__[key] = value

class Person(AttrMixin):
    "Represents a person"
    def __init__(self, name):
        self.name = name
```

Note In object-oriented languages, a *mixin* class is a class that contains a certain behavior to be inherited by subclasses to add specific behaviors. A class can inherit some or all of its behaviors from one or more mixins. Ending the name with "Mixin" is not required; it's simply a flag so that your behavior-focused classes are clearly delineated.

Here the **Person** class inherits its attribute setting behavior from the AttrMixin class. The **AttrMixin.__setattr__()** method does here make the definite assumption that the classes it will be mixed in with are storing all attributes using the standard instance __dict__ mechanism. When you start to see the layers of behavior that Python allows you to add to the process of attribute assignment, you will realize that this may be a dangerous assumption, but certainly it holds for the **Person** class.

As usual, you can verify the actions of this code in the interactive interpreter. Note that the setting of an attribute is reported whether it is set inside an object method or in external code.

__getattr__()

Attribute retrieval works a little differently from attribute setting. When you try to access an attribute of some instance *o*, the interpreter looks in o.__dict__; if the attribute is not found there, it looks in the instance's class, then in that class's superclass, and so on. *Only if the attribute is not found does the interpreter then call the instance's* ***__getattr__()*** *method with the name of the attribute*. It is conventional for **__getattr__()** to raise the AttributeError exception when the attribute name provided is unacceptable for some reason.

CODE TO EDIT: attrmagic.py

```
"""
Demonstrate magic methods for attribute access.
"""
class AttrMixin:
    "Displays a message when an object's attributes are retrieved or set."

    def __setattr__(self, key, value):
        print("ATTR: setting attribute {0!r} to {1!r}".format(key, value))
        self.__dict__[key] = value

    def __getattr__(self, key):
        print("ATTR: getting attribute {0!r}".format(key))
        self.__setattr__(key, "No value")
        return "No value"

class Person(AttrMixin):
    "Represents a person"
    def __init__(self, name):
        self.name = name
```

Start an entirely new interactive console session in which to test the updated module—remember, the code of a module is executed only on the first import. Trying to import the updated module will therefore fail, and you will not see the expected behaviors.

Note There are ways to trigger re-import of a module without restarting the interactive interpreter. The Ellipse teaching system surrounds your interactive console and provides a subtly different environment from the classic interactive console, and since Ellipse makes it so easy, we have you start new interactive sessions. This ensures that you are starting with a pristine environment, to ensure that you get the same results we observed and recorded during course production.

CODE TO TYPE: Enter the following code in an interactive console session

```
>>> from attrmagic import *
>>> steve = Person("Steve Holden")
ATTR: setting attribute 'name' to 'Steve Holden'
>>> steve.newattr
ATTR: getting attribute 'newattr'
ATTR: setting attribute 'newattr' to 'No value'
'No value'
>>> steve.newattr
'No value'
>>> steve.name
'Steve Holden'
>>>
```

Observe that while the first access to the **newattr** attribute results in a call to **__getattr__()**, the second one does not. This is because the first call actually sets a value in the object's __dict__ and so the second attempt finds the attribute using the standard methods.

Note The interpreter uses a slightly different mechanism to access the special attributes whose names begin and end in double underscores. This is done to enforce certain standard object behaviors, which otherwise could be overridden.

__delattr__()

This method is called whenever an attribute is deleted from an object. Again, we'll publish a message to show what can be done with this method.

CODE TO EDIT: attrmagic.py

```
"""
Demonstrate magic methods for attribute access.
"""
class AttrMixin:
    "Displays a message when an object's attributes are retrieved, deleted or se
t."

    def __setattr__(self, key, value):
        print("ATTR: setting attribute {0!r} to {1!r}".format(key, value))
        self.__dict__[key] = value

    def __getattr__(self, key):
        print("ATTR: getting attribute {0!r}".format(key))
        self.__setattr__(key, "No value")
        return "No value"

    def __delattr__(self, key):
        print("ATTR: Deleting key {0!r}".format(key))
        object.__delattr__(self, key)

class Person(AttrMixin):
    "Represents a person"
    def __init__(self, name):
        self.name = name
```

Note that your version of __delattr__ simply *delegates* the deletion to Python's standard **object** behavior. This allows you to ignore whatever complexities may be required in deletion. Again, test your modifications:

CODE TO TYPE: Test your modifications in a new interactive interpreter console

```
>>> from attrmagic import *
>>> student = Person("your name")
ATTR: setting attribute 'name' to 'your name'
>>> student.age
ATTR: getting attribute 'age'
ATTR: setting attribute 'age' to 'No value'
'No value'
>>> student.age = 21
ATTR: setting attribute 'age' to 21
>>> student.age
21
>>> del student.age
ATTR: Deleting key 'age'
>>> student.age
ATTR: getting attribute 'age'
ATTR: setting attribute 'age' to 'No value'
'No value'
>>> del student.__delattr__
ATTR: Deleting key '__delattr__'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "attrmagic.py", line 18, in __delattr__
    object.__delattr__(self, key)
AttributeError: __delattr__
>>>
```

You can see that the attempt to delete __delattr__ fails, because the special attributes are not discovered in the same way as the regular ones. This avoids the deletion of standard behaviors that are required to be true of all objects.

Emulating Functions: the __call__() Method

Implementing the **__call__()** method allows you to make your instances callable, just as though they were regular functions. Create **callmagic.py** in your **Python3_Lesson09/src** folder as shown:

CODE TO TYPE: callmagic.py

```
"""
Demonstrate how to make instances callable.
"""

class funclike:
    def __call__(self, *args, **kwargs):
        print("Args are:", args)
        print("Kwargs are:", kwargs)

f = funclike()
f(1, 2, 3, this="one", that="the other")
```

Save and run it:

OBSERVE: Result of running callmagic.py

```
Args are: (1, 2, 3)
Kwargs are: {'this': 'one', 'that': 'the other'}
```

In this chapter, we've started to investigate the relationship between the interpreter and the objects that we create. This explanation should make you more aware of what is going on "under the hood," and give you some idea of the wider possibilities for using Python to solve your problems. Most of the time the standard interpreter behavior is perfectly acceptable. For those occasions when it is not, you now have some idea how to modify it.

There are other special methods that we have not covered yet, but for now you have done quite enough. Take a break, and then move on to the next lesson, where you will be learning how to make your objects' behaviors even more complex while retaining the essential simplicity of Python.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Properties

A property is a special sort of class attribute. You access it like a standard attribute, but "under the hood," the interpreter runs methods ("getters" to access the data and "setters" to store new data) to produce the required results. The data-like syntax is easier to read and write than lots of method calls, yet the interposition of the method calls allows for data validation, active updating, and/or read-only attributes. Before looking in detail at properties, you should understand some of the reasons they are desirable.

Putting Computations Behind Attributes

In the last lesson, we learned about several special methods that let us access and control attributes—**__getattr__()**, **__setattr__()**, and **__delattr__()**. You can use these techniques to control the value of various attributes—but remember that the **__getattr__()** method will only be used if normal attribute access fails to find the named attribute. Therefore, you'll want to store the values of "managed" attributes (values that must be processed on retrieval) in a special directory, to ensure that normal attribute access does not find them. The following code sample demonstrates control of specific attributes via the **__getattr__()** method.

Suppose you want to keep a first name, last name, age, list of classes, and a grade for teachers in a school. Further suppose that you were prepared to allow some laxity in data entry, but that you always wanted to return the names properly capitalized, the age as an integer, the list of classes in sorted order and the grade as a string (though it should be entered as a number). This kind of management is precisely what the attribute-handling special methods were designed for.

As is usually the case, there must be test code, which follows first in the time-honored tradition of TDD—Test-Driven Development. Create a **Python3_Lesson10** project and assign it to the **Python3_Lessons** working set. Then, create **test_teacher.py** in your **Python3_Lesson10/src** folder as shown:

test_teacher.py

```
import unittest
from teacher import Teacher

class TestTeacher(unittest.TestCase):

    def setUp(self):
        self.teacher = Teacher("steve",
                               "holden",
                               "63",
                               ["Python 3-3","Python 3-1","Python 3-2"],
                               5)

    def test_get(self):
        self.assertEqual(self.teacher.first_name, "Steve")
        self.assertEqual(self.teacher.last_name, "Holden")
        self.assertEqual(self.teacher.age, 63)
        self.assertEqual(self.teacher.classes, ["Python 3-3","Python 3-1","Python 3-2"]
)
        self.assertEqual(self.teacher.grade, "Fifth")
        self.teacher.description = "curmudgeon"
        self.assertEqual(self.teacher.description, "curmudgeon")
if __name__ == "__main__":
    unittest.main()
```

We'll start out with a simplistic implementation of the **Teacher** class that simply stores the attributes as regular values and relies on the standard Python mechanism for attribute retrieval. Since no transformation is taking place on the data, it should not be too surprising if this first naive implementation fails. Create **teacher.py** in the same folder as shown:

CODE TO TYPE: teacher.py

```
class Teacher(object):

    grades = {1: "First", 2: "Second", 3: "Third", 4: "Fourth", 5: "Fifth"}

    def __init__(self, first_name, last_name, age, classes, grade):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.classes = classes
        self.grade = grade
```

Save both files and run **test_teacher.py**. Sure enough, you see a failure:

OBSERVE: The simple implementation fails

```
F
======================================================================
FAIL: test_get (__main__.TestTeacher)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\sholden\workspace\Python3_Lesson8\src\test_teacher.py", line 14, in te
st_get
    self.assertEqual(self.teacher.first_name, "Steve")
AssertionError: 'steve' != 'Steve'

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

One of the beauties of Python, however, is that it is almost infinitely flexible, and so it is quite possible to change this implementation to do what is required. Although it may seem contradictory, the first thing you need to change is the way the object stores attributes—until you do that, the attribute assignments will always result in their being available without invoking **__getattr__()**.

CODE TO EDIT: teacher.py

```
"""
Demonstrate simple attribute management
"""
class Teacher(object):

    grades = {1: "First", 2: "Second", 3: "Third", 4: "Fourth", 5: "Fifth"}

    def __init__(self, first_name, last_name, age, classes, grade):
        self.__dict__['_attrs'] = {}
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.classes = classes
        self.grade = grade

    def __setattr__(self, name, value):
        self._attrs[name] = value

    def __getattr__(self, name):
        if name not in self._attrs:
            raise AttributeError("Teacher has no attribute {0!r}".format(name))
        value = self._attrs[name]
        if name in ("first_name", "last_name"):
            return value.capitalize()
        elif name == "age":
            return int(value)
        elif name == "classes":
            return sorted(value)
        elif name == "grade":
            return self.grades[value]
        else:
            return value
```

Here the **__init__** method creates a regular attribute called *_attrs*, a dict in which the attribute values are kept, by making a direct entry in the instance's __dict__. It uses this technique to avoid a direct assignment, which would invoke the instance's **__setattr__()** method. That method attempts to store the attribute value against its name **self._attrs**, which would need to be looked up by **__getattr__()**. This in turn would try and find the name "_attrs" in the **self._attrs** dict, which would again invoke **__getattr__()**, and so on. This infinite regression would only terminate when the interpreter ran out of *stack*, the area of memory where it stores partially-completed function namespaces.

Note

The convention in Python is that attributes whose names begin with "_" are internal to the implementation of a class. Because of that, such attributes don't appear in **help** but do appear in the output from **dir()**. While there is nothing to stop you from accessing these attributes directly, the naming convention acts as a flag that outside interference is likely to break the internal logic.

Now, all attributes are stored in the **_attrs** dict, and the **__getattr__()** method uses the name of the retrieved attribute to decide what processing needs to be performed on the stored value in order to meet specifications. Save both files and run **test_teacher.py**. Happily, the updated object should now pass its tests.

OBSERVE: All tests now pass

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

This may not look so bad at a glance, but maintenance for this code is challenging. For example, if you wanted to create a subclass that had different behavior on just the "age" attribute, you would have to rewrite the **__getattr__()** method for the child class. Then if the parent had a bug, you might have to rewrite both the parent and child. As you might imagine, this quickly leads to fragile code, and tends to encourage code duplication (which is normally held to be a bad thing).

For example, suppose you want to create a Teacher subclass that supports gender differences. If male, the Teacher subclass returns "Mr." at the start of "first_name." If female, it returns "Ms." The current design forces you to completely rewrite the **__getattr__()** method, because it is "monolithic"—all the attributes are dealt with in the same method, so changing the response for just one attribute is difficult or impossible.

An alternative is to use *properties*. Properties let you assign computations to accesses involving a specific attribute, so if you inherit the class, you can easily extend it without having to dance around the subtleties of **__getattr__()**. This allows you to easily change one small method without worrying about tangling with a multitude of unrelated attributes.

A Teacher Class Constructed of Properties

A *property* in Python is a data component to which access is mediated by methods, even though the user of the property can treat it as a simple data attribute. This allows you to hide a layer of logic underneath attribute-style access to an object's data.

Note

If you *know* in advance that the logic is required, there is something to recommend simply writing the methods and documenting them as the necessary solution to the problem. Properties excel when the logic needs to be introduced later, after you have already written code that treats the data as simple attributes. Under those circumstances, properties allow you to insert a layer of logic without changing the code that currently uses the attributes.

Like a lot of other programming descriptions, this sounds a lot more complex than it is. And since a bit of code often helps to clarify new concepts, let's construct the teacher class with properties using the techniques as we've described:

CODE TO EDIT: teacher.py

```
class Teacher(object):

    grades = {1: "First", 2: "Second", 3: "Third", 4: "Fourth", 5: "Fifth"}

    def __init__(self, first_name, last_name, age, classes, grade):
        ~~self.__dict__['_attrs'] = {}~~
        ~~self.first_name = first_name~~
        ~~self.last_name = last_name~~
        ~~self.age = age~~
        ~~self.classes = classes~~
        ~~self.grade = grade~~
        self._first_name = first_name # internal data attributes are set
        self._last_name = last_name
        self._age = age
        self._classes = classes
        self._grade = grade

    ~~def __setattr__(self, name, value):~~
        ~~self._attrs[name] = value~~

    ~~def __getattr__(self, name):~~
        ~~if name not in self._attrs:~~
            ~~raise AttributeError("Teacher has no attribute {0!r}".format(name))~~
        ~~value = self._attrs[name]~~
        ~~if name in ("first_name", "last_name"):~~
            ~~return value.capitalize()~~
        ~~elif name == "age":~~
            ~~return int(value)~~
        ~~elif name == "classes":~~
            ~~return sorted(value)~~
        ~~elif name == "grade":~~
            ~~return self.grades[value]~~
        ~~else:~~
            ~~return value~~

    def first_name(self):
        return self._first_name.capitalize()
    first_name = property(first_name)

    def last_name(self):
        return self._last_name.capitalize()
    last_name = property(last_name)

    def age(self):
        return int(self._age)
    age = property(age)

    def classes(self):
        return sorted(self._classes)
    classes = property(classes)

    def grade(self):
        return self.grades[self._grade]
    grade = property(grade)
```

Save both files and run **test_teacher.py**.

running test_teacher.py with teacher.py changed to use properties

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Thanks to the magic of **unittest**, this demonstration of a new programming technique appears to be a valid refactoring. At least you have passed a definite "smoke test" by passing the current tests. Let's review the changes in the code of teacher.py:

OBSERVE: teacher.py __init__() method

```
def __init__(self, first_name, last_name, age, classes, grade):
    self._first_name = first_name # internal data attributes are set
    self._last_name = last_name
    self._age = age
    self._classes = classes
    self._grade = grade
```

In the __init__() method, we set **first_name** via **self._first_name**. This is done to provide a data attribute on which to base the **first_name** property (if it had the same name as the property, the assignment would overwrite the method!). We made similar changes for the other managed attributes.

OBSERVE: New Methods in teacher.py

```
def first_name(self):
    return self._first_name.capitalize()
first_name = property(first_name)

def last_name(self):
    return self._last_name.capitalize()
last_name = property(last_name)

def age(self):
    return int(self._age)
age = property(age)

def classes(self):
    return sorted(self._classes)
classes = property(classes)

def grade(self):
    return self.grades[self._grade]
grade = property(grade)
```

The **first_name()** method accesses the **_first_name** data attribute, and processes it before returning it as the value of the attribute. We provide similar methods for the other managed attributes.

The **first_name()** method becomes a property when it is replaced inside the class body by the result of calling the built-in **property()** function with the method as an argument. We changed the other managed attributes likewise into properties.

You can see that if you wanted to create a Teacher subclass where the first_name attribute was modified by a gender attribute, you would only need to redefine the first_name property in your subclass—the other property definitions would continue to stand. This is in distinction to the preceding class, whose "monolithic" (all in one piece) **__getattr__()** makes it hard to separate one attribute from another.

Decorator Syntax

Because defining a function or method and then applying a function such as **property()** to it is a common pattern, Python has a special shorthand for it. The syntax we used above was:

Standard Property Creation

```
def method(self, ...):
    """Method body."""
    ...
method = property(method)
```

This application of a function to another function is called *decoration*, and the applied function (in this case **property**) is called a *decorator*. If the method is lengthy, the final reassignment to the method name is easy to miss. Consequently you can also use the following syntax, which is merely a shortcut for the standard mechanism above:

Property Creation with a Decorator

```
@property
def method(self, ...):
    """Method body."""
    ...
```

You should find that the code works exactly the same using this syntax as it does using the standard property creation. Try it, just to be sure.

CODE TO EDIT: teacher.py

```
class Teacher(object):

    grades = {1: "First", 2: "Second", 3: "Third", 4: "Fourth", 5: "Fifth"}

    def __init__(self, first_name, last_name, age, classes, grade):
        self._first_name = first_name # internal data attributes are set
        self._last_name = last_name
        self._age = age
        self._classes = classes
        self._grade = grade

    @property
    def first_name(self):
        return self._first_name.capitalize()
    ~~first_name = property(first_name)~~

    @property
    def last_name(self):
        return self._last_name.capitalize()
    ~~last_name = property(last_name)~~

    @property
    def age(self):
        return int(self._age)
    ~~age = property(age)~~

    @property
    def classes(self):
        return sorted(self._classes)
    ~~classes = property(classes)~~

    @property
    def grade(self):
        return self.grades[self._grade]
    ~~grade = property(grade)~~
```

As always, the first thing that you should do after changing your code is... run your tests! These two ways to apply properties to methods are entirely equivalent, so your tests should continue to pass.

Settable Properties

Note that while the first implementation correctly allowed you to reassign the managed attributes through use of the **__setattr__()** method, this one does not. Neither can you delete managed attributes (which was also an issue with the earlier code, though we did not mention it at the time. You can verify this using an interactive interpreter session:

CODE TO TYPE: Verify

```
>>> from teacher import *
>>> t = Teacher("steve", "holden", "63",
...                ["Python 3-3","Python 3-1","Python 3-2"], 5)
>>>
>>> t.first_name
'Steve'
>>> t.first_name = "joe"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> del t.first_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>>
```

So at the moment you can neither assign to nor delete the managed attributes. You will be changing the tests to include these features shortly, to provide failing tests that new functionality in your **teacher** module can turn into success.

The built-in **property** function is actually rather more complicated than you have so far seen. Its full *signature* (the pattern of arguments it can be called with) is as follows.

property function signature

```
property(fget=None, fset=None, fdel=None, doc=None)
```

fget is the *getter function*, **fset** is the *setter function*, **fdel** is the *deleter function* and **doc** is the *documentation*. So the reason that the properties you have defined so far cannot be set is that the *decorator syntax only passes a single argument to the call of* ***property()***. This single positional argument is associated (positionally) with the **fget** parameter, so you can get the attribute value, but there is no way to set or delete the attributes (and no documentation!)

Setting values via properties

Properties do more than just provide the ability to compute values during retrieval of attributes. They also let you perform calculations while *setting* values. This is useful during validation of incoming data. For example, what if you wanted to confirm that the age attribute was passed a valid integer rather than converting it to an integer when it was accessed? Using standard techniques, you would declare a second method and pass it as the second argument to the call of **property()**. First, of course, we need to add a new test to the test suite that fails. This should be fairly easy with the experience we had in the interactive interpreter session above. Since we want the values we can assign to age to be limited to integers, we'll also add a test to make sure that any other type of data raises a ValueError exception.

CODE TO EDIT: test_teacher.py

```
import unittest
from teacher import Teacher

class TestTeacher(unittest.TestCase):

    def setUp(self):
        self.teacher = Teacher("steve",
                                "holden",
                                "63",
                                ["Python 3-3","Python 3-1","Python 3-2"],
                                5)

    def test_get(self):
        self.assertEqual(self.teacher.first_name, "Steve")
        self.assertEqual(self.teacher.last_name, "Holden")
        self.assertEqual(self.teacher.age, 63)
        self.assertEqual(self.teacher.classes, ["Python 3-1","Python 3-2","Pytho
n 3-3"])
        self.assertEqual(self.teacher.grade, "Fifth")

        self.teacher.description = "curmudgeon"
        self.assertEqual(self.teacher.description, "curmudgeon")

    def test_set(self):
        self.teacher.age = "21"
        self.assertEqual(self.teacher._age, 21)
        self.assertEqual(self.teacher.age, 21)
        self.assertRaises(ValueError, self.setAgeWrong)

    def setAgeWrong(self):
        self.teacher.age = "twentyone"

if __name__ == "__main__":
    unittest.main()
```

Note that **unittest.TestCase.assertRaises** expects a function and an exception as arguments. It calls the function, and flags a failure if the call does not raise the specified exception type. After these modifications, you would expect your new test to fail, and sure enough it does (before it even gets around to testing to see whether the **setAgeWrong** method raises the required exception).

Tests fail as expected before you update your teacher class

```
.E
======================================================================
ERROR: test_set (__main__.TestTeacher)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\sholden\workspace\Python3_Lesson8\src\test_teacher2.py", line 2
3, in test_set
    self.teacher.age = 21
AttributeError: can't set attribute

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (errors=1)
```

Having updated the tests to make it obvious that an upgrade is required to the teacher module, we need to add the necessary new code. We'll start by using the standard method to give the **age** attribute both a getter *and* a setter.

CODE TO EDIT: teacher.py

```
class Teacher(object):

    grades = {1: "First", 2: "Second", 3: "Third", 4: "Fourth", 5: "Fifth"}

    def __init__(self, first_name, last_name, age, classes, grade):
        self._first_name = first_name # internal data attributes are set
        self._last_name = last_name
        ~~self._age = age~~
        self.age = age
        self._classes = classes
        self._grade = grade

    @property
    def first_name(self):
        return self._first_name.capitalize()

    @property
    def last_name(self):
        return self._last_name.capitalize()

    ~~@property~~
    ~~def age(self):~~
        ~~return int(self._age)~~

    def getage(self):
        return self._age

    def setage(self, value):
        self._age = int(value)

    age = property(getage, setage, doc="Teacher's age: must be convertible to in
teger")

    @property
    def classes(self):
        return sorted(self._classes)

    @property
    def grade(self):
        return self.grades[self._grade]
```

Note You will see that the **__init__()** method is now relying on the property to establish the initial value of the managed attribute rather than directly assigning to the underlying data member. This is generally a good thing, since if the setter method performs validations these will also be applied to the initial value passed in as an argument to **__init__()**.

Now the **age** attribute has both a getter and a setter, you should see that it passes all tests with flying colors.

OBSERVE: All tests pass once teacher.py is modified

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

You may wonder whether it is possible to achieve the same ends using decorators, and the answer is yes. This is because a read-only property (which, you will remember, can be created with the use of a decorator because it only requires a single argument) has a **setter()** method that can be used to decorate a... setter method! This means that you can create the age property as before, with a decorator, and then decorate the setter() method with a method of the getter() property that you just created.

This may sound a little confusing, but once you have typed the code, it should seem a little more natural. The

age() property goes back to its original code, and the age setter is decorated by one of the getter property's methods (the getter has been defined specifically to provide these extra methods as a convenience).

CODE TO EDIT: teacher.py

```
class Teacher(object):

    grades = {1: "First", 2: "Second", 3: "Third", 4: "Fourth", 5: "Fifth"}

    def __init__(self, first_name, last_name, age, classes, grade):
        self._first_name = first_name # internal data attributes are set
        self._last_name = last_name
        self.age = age
        self._classes = classes
        self._grade = grade

    @property
    def first_name(self):
        return self._first_name.capitalize()

    @property
    def last_name(self):
        return self._last_name.capitalize()

    ~~def getage(self):~~
    @property
    def age(self):
        return self._age

    ~~def setage(self, value):~~
    @age.setter
    def age(self, value):
        self._age = int(value)

    ~~age = property(getage, setage, doc="Teacher's age: must be convertible to integer")~~

    @property
    def classes(self):
        return sorted(self._classes)

    @property
    def grade(self):
        return self.grades[self._grade]
```

Note The second age definition might be flagged as a "duplicate signature" in Eclipse; you can safely ignore this for now.

You should, of course, confirm as usual that your tests continue to succeed.

Deleting Attributes Using Properties

Deleting attributes works in nearly the same fashion as setting attributes. Create yet another function with the same name as your attribute and place a **@<my-attribute-name>.deleter**. In our next example, removing a grade means creating a grade function, placing a **@grade.deleter** above it, and then in the logic, adding a year to the age of the teacher.

First, let's write a test for our expected behavior:

CODE TO EDIT: test_teacher.py

```
import unittest
from teacher import Teacher

class TestTeacher(unittest.TestCase):

    def setUp(self):
        self.teacher = Teacher("steve",
                               "holden",
                               "63",
                               ["Python 3-3","Python 3-1","Python 3-2"],
                               5)

    def test_get(self):
        self.assertEqual(self.teacher.first_name, "Steve")
        self.assertEqual(self.teacher.last_name, "Holden")
        self.assertEqual(self.teacher.age, 63)
        self.assertEqual(self.teacher.classes, ["Python 3-3","Python 3-1","Pytho
n 3-2"])
        self.assertEqual(self.teacher.grade, "Fifth")

        self.teacher.description = "curmudgeon"
        self.assertEqual(self.teacher.description, "curmudgeon")

    def test_set(self):
        self.teacher.age = "21"
        self.assertEqual(self.teacher._age, 21)
        self.assertEqual(self.teacher.age, 21)
        self.assertRaises(ValueError, self.setAgeWrong)

    def setAgeWrong(self):
        self.teacher.age = "twentyone"

    def test_delete(self):
        del self.teacher.grade
        self.assertEqual(self.teacher.age, 64)
        self.assertRaises(AttributeError, self.accessGrade)

    def accessGrade(self):
        return self.teacher.grade

if __name__ == "__main__":
    unittest.main()
```

As usual, a newly added test should fail. You should verify this, as usual, by running the updated test suite.

OBSERVE: The deletion tests fail, as expected

```
E..
======================================================================
ERROR: test_delete (__main__.TestTeacher)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\sholden\workspace\Python3_Lesson8\src\test_teacher3.py", line 3
2, in test_delete
    del self.teacher.grade
AttributeError: can't delete attribute

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (errors=1)
```

Now, modify the teacher.py code to add the deleter method:

CODE TO EDIT: teacher.py

```
class Teacher(object):

    grades = {1: "First", 2: "Second", 3: "Third", 4: "Fourth", 5: "Fifth"}

    def __init__(self, first_name, last_name, age, classes, grade):
        self._first_name = first_name
        self._last_name = last_name
        self._age = age
        self._classes = classes
        self._grade = grade

    @property
    def first_name(self):
        return self._first_name.capitalize()

    @property
    def last_name(self):
        return self._last_name.capitalize()

    @property
    def age(self):
        return int(self._age)

    @age.setter
    def age(self, value):
        self._age = int(value)

    @property
    def classes(self):
        return sorted(self._classes)

    @property
    def grade(self):
        return self.grades[self._grade]

    @grade.setter
    def grade(self, value):
        self.grades[value] # throw error if value != a key
        self._grade = value

    @grade.deleter
    def grade(self):
        self.age += 1
        del self._grade
```

Note The updated "age" property now applies the int() built-in to its argument. This allows users to specify the age as a character string without the code breaking. Whether this is a good idea or not, and whether the setter should even accept strings or not, is an interesting question—one that we will ignore for now.

Save it and run the test. All tests should pass immediately.

OBSERVE: The newly-added deleter tests now pass

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

So you now understand how you can put logic behind all types of attribute access. Beware of using this technique when it isn't really necessary: remember, if you know method calls are going to be required from the outset, it is much better to build them into the API for your objects from the start. If the logic needs to be retrofitted, however, properties are a really useful way of fitting it.

Properties allow you to isolate each piece of logic in its own method, making it easy to extend and reuse as a parent superclass or to implement in child classes. They are often used for validation, logging, formatting, and a myriad of other tasks. The only possible downside to properties is that they require a little bit of extra work, but the extra functionality they provide is generally well worth that effort.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

A First Look at Logging

When you want to save data about a program's operation (typically to record the actions of your program, or particular error conditions that have occurred) you have a number of choices. Informal results can be printed to the standard output stream, but the only person who will see this output is the user, and once the window is closed the output is lost (there is also a standard error stream, with the same disadvantages). You could also write information to a file. This can work for a program, but it is difficult to use as part of a module that might be used in many different circumstances: ideally the program will log all output to the same destination, but how can you make an unrelated collection of modules do that?

Furthermore, it would be nice to be able to store information during debugging, and then be able to suppress the debug output when the program goes into production. Ideally, you'd like to do this without having to edit the code to remove or comment out the debugging output statements, and the debug output would be cleanly separated from the program's normal output. Then the stored output could form a long-term information stream that allows you to examine your program's performance over its entire lifespan.

Finally and perhaps most importantly, you want to be able to share your code! What if you need to capture the progress and mistakes of others using your work? Yes, you could do this by writing user actions to a file, but then you run into the danger of making your code somewhat confusing—especially if your program relies on file output for real tasks such as saving important files.

This is where *logging* comes to the rescue! This is not the process in which certain trees are cut down by a lumberjack, but rather a process whereby data is stored over time in such a way as to be as unobtrusive to the operating software as possible. Due to a certain lack of imagination, however, the programming examples will involve lumberjacks in the best Monty Python tradition.

Fortunately the standard library contains the solution for all of these issues in the logging module. It is easy to use and flexible in operation. There's a lot to learn!

This lesson includes the following sections:

- Setting Up a Basic Logger
- Other Logging Functions
- Other Logging Levels
- Getting Tests to Use Different Logging Levels
- Log Formatting

Setting Up a Basic Logger

To set up a basic logger, you import the **logging** module, call its **basicConfig()** utility function, and then start logging. First, create a **Python3_Lesson11** project as usual and assign it to the **Python3_Lessons** working set, so we have a place to store the files. Then type the code shown below.

CODE TO TYPE: Create a log from the interactive console

```
>>> import logging
>>> logging.basicConfig(filename='V:/workspace/Python3_Lesson11/src/output.log',level=logging.DEBUG)
>>> logging.debug('My first log entry!')
```

This creates a file in your **Python3_Lesson11/src** folder named **output.log**. Let's take a look at the contents.

contents of output.log

```
DEBUG:root:My first log entry!
```

This is pretty handy, but doesn't really showcase how useful logging is. So let's create a slightly more sophisticated example representing lumberjacks cutting down trees. A Lumberjack starts with no tree. After you assign a Tree object to the Lumberjack, he can chop it down, which turns it into a number of boards (determined by the size of the tree), and then you remove the tree from the Lumberjack object.

The basic API for a Tree is pretty simple. You create it by calling **Tree(s)** where s is a *size code*—one of "S," "M," "L," "XL," or "XXL". Instances have a **get_boards()** method that you call to learn the number of boards the tree can produce (1 for a size "S" tree, 5 for a size "XXL"). Trees represent themselves as "Tree: Size S" or similar.

The Lumberjack is not that much more complicated in its initial implementation. Created by calling the class **Lumberjack()**, each instance starts out with no tree. Once a tree is assigned it can be cut down and converted into boards by calling the Lumberjack's **cut_down_tree()** method. If this method is called when the Lumberjack has no tree, a TypeError exception is raised.

As usual, we'll start by writing basic tests for the Tree and Lumberjack classes. We test the Trees in a number of ways: for each size of tree, **test_lumber()** verifies that the tree size returns the expected number of boards. **test_string()** verifies that the Tree objects do represent themselves as required, and **test_code()** verifies that an exception is raised when the class is called with an invalid size code. You test the lumberjack by creating a new one for each size of tree, verifying there is initially no tree, assigning a tree, cutting it down and verifying that the Lumberjack no longer has a tree and that the right number of boards were produced. In your **Python3_Lesson11/src** folder, create **test_forestry.py** as shown:

CODE TO TYPE: test_forestry.py

```
import unittest

from forestry import Lumberjack, Tree

sizes = (("S", 1), ("M", 2), ("L", 3), ("XL", 4), ("XXL", 5))

class TestTree(unittest.TestCase):

    def test_lumber(self):
        for code, boards in sizes:
            tree = Tree(code)
            self.assertEqual(boards, tree.get_boards())

    def test_string(self):
        tree = Tree("L")
        self.assertEqual(str(tree), "Tree: Size L")

    def test_exceptions(self):
        self.assertRaises(ValueError, Tree, "parrot")
        self.assertRaises(TypeError, Lumberjack().cut_down_tree)

class TestLumberjack(unittest.TestCase):

    def test_lumberjack(self):
        for code, boards in sizes:
            tree = Tree(code)
            graham = Lumberjack()
            self.assertIsNone(graham.tree)
            graham.tree = tree
            brds = graham.cut_down_tree()
            self.assertIsNone(graham.tree)
            self.assertEqual(boards, brds)

if __name__ == "__main__":
    unittest.main()
```

If you are getting the hang of test-driven development, you're already thinking about what your Tree and Lumberjack classes need to do to pass these tests, but you should start with the "simplest possible thing that can fail" first and verify that the tests do actually fail or give errors.

In your **Python3_Lesson11/src** folder, create **forestry.py** as shown

CODE TO TYPE: forestry.py

```
class Tree(object):
    "Represent a tree in a forest that can be converted into boards."
    sizes = dict(S=1, M=2, L=3, XL=4, XXL=5)

    def __init__(self, size="L"):
        "Initialize."
        self.size = size

    def get_boards(self):
        "Return number of boards equivalent."
        return self.sizes[self.size]

    def __str__(self):
        "Render as a string."
        return "Tree: Size %s" % self.size

class Lumberjack(object):
    "Represent a lumberjack who can cut down trees."

    def cut_down_tree(self):
        "Convert tree to boards and go back to not having a tree."
        pass
```

When you run the test with this vestigial implementation, you will not surprisingly find that the tests don't all pass (but note that some do, because Tree correctly implements both **get_boards()** and **__str__()**).

OBSERVE: Not all tests pass—that's OK!

```
EF..
======================================================================
ERROR: test_lumberjack (__main__.TestLumberjack)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson11\src\test_forestry.py", line 28, in test_lumberjac
k
    self.assertIsNone(graham.tree)
AttributeError: 'Lumberjack' object has no attribute 'tree'

======================================================================
FAIL: test_exceptions (__main__.TestTree)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python3_Lesson11\src\test_forestry.py", line 19, in test_exception
s
    self.assertRaises(ValueError, Tree, "parrot")
AssertionError: ValueError not raised by Tree

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=1, errors=1)
```

The **test_lumberjack()** fails because the test assumes that a newly created Lumberjack object will have a **tree** attribute with the value **None**. This is easily arranged in its **__init__()** method. **test_exceptions()** fails because the **__init__()** method is not validating the size argument. This is again fairly easily added. Make the necessary changes and ensure that then all four tests pass.

CODE TO EDIT: forestry.py

```
class Tree(object):
    "Represent a tree in a forest that can be converted into boards."
    sizes = dict(S=1, M=2, L=3, XL=4, XXL=5)

    def __init__(self, size="L"):
        "Initialize: insist that size is a valid code."
        if size not in self.sizes:
            message = "Tree size must be one of: %s" % ",".join(self.sizes.keys())
            raise ValueError(message)
        self.size = size

    def get_boards(self):
        "Return number of boards equivalent."
        return self.sizes[self.size]

    def __str__(self):
        "Render as a string."
        return "Tree: Size %s" % self.size

class Lumberjack(object):
    "Represent a lumberjack who can cut down trees."
    def __init__(self):
        "Initialize: start with no tree."
        self.tree = None

    def cut_down_tree(self):
        "Convert tree to boards and go back to not having a tree."
        ~~pass~~
        if not self.tree:
            raise TypeError("Cannot cut_down_tree(): Lumberjack has no tree!")
        boards = self.tree.get_boards()
        self.tree = None
        return boards

if __name__ == "__main__":
    "Demonstrate basic usage."
    john = Lumberjack()
    john.tree = Tree("XXL")
    if john.cut_down_tree() != 5:
        print("Error: XXL tree should yield 5 boards")
```

Save it and run the test again. All tests should pass now, and we can add in a simple logger. This just involves adding a few lines at the beginning of the module.

CODE TO EDIT: forestry.py

```
# import the logging module
import logging

# set up the logger
logging.basicConfig(filename='forestry.log',level=logging.DEBUG)

# log a message
logging.info('Starting up the forestry program')

class Tree(object):
    "Represent a tree in a forest that can be converted into boards."
    sizes = dict(S=1, M=2, L=3, XL=4, XXL=5)

    def __init__(self, size="L"):
        "Initialize: insist that size is a valid code."
        if size not in self.sizes:
            message = "Tree size must be one of: %s" % ",".join(self.sizes.keys())
            raise ValueError(message)
        self.size = size

    def get_boards(self):
        "Return number of boards equivalent."
        return self.sizes[self.size]

    def __str__(self):
        "Render as a string."
        return "Tree: Size %s" % self.size

class Lumberjack(object):
    "Represent a lumberjack who can cut down trees."
    def __init__(self):
        "Initialize: start with no tree."
        self.tree = None

    def cut_down_tree(self):
        "Convert tree to boards and go back to not having a tree."
        boards = self.tree.get_boards()
        self.tree = None
        return boards

if __name__ == "__main__":
    "Demonstrate basic usage."
    john = Lumberjack()
    john.tree = Tree("XXL")
    if john.cut_down_tree() != 5:
        print("Error: XXL tree should yield 5 boards")
```

Run both test_forestry.py and forestry.py. The tests should continue to pass, and the forestry program should run without errors or any output. In the **Python3_Lesson11/src** folder, you'll see a new **forestry.log** file. Open it and you should see:

contents of forestry.log

```
INFO:root:Starting up the forestry program
INFO:root:Starting up the forestry program
```

Look familiar? But why are there two entries? There are two entries because you loaded forestry.py twice, once when you ran it by itself and the other time in test_forestry.py, thanks to the line **from forestry import Lumberjack, Tree**. Also, because the logging system records things over time, each time it is called it appends to the existing file. This means that your log files are a living history of your application (though that history is of somewhat limited interest right now due to the restricted information that appears in it). But, every time your module is used, it logs that fact in the log file!

Now let's make it a little more interesting. Sprinkle some log messages throughout the **forestry.py** code:

CODE TO EDIT: forestry.py

```
import logging

# set up the logger
logging.basicConfig(filename='forestry.log',level=logging.DEBUG)

# log a message
logging.info('Starting up the forestry program')

class Tree(object):
    "Represent a tree in a forest that can be converted into boards."
    sizes = dict(S=1, M=2, L=3, XL=4, XXL=5)

    def __init__(self, size="L"):
        "Initialize: insist that size is a valid code."
        if size not in self.sizes:
            message = "Tree size must be one of: %s" % ",".join(self.sizes.keys())
            raise ValueError(message)
        self.size = size
        logging.info('Instantiated a tree')

    def get_boards(self):
        "Return number of boards equivalent."
        logging.info('tree.get_boards method called')
        return self.sizes[self.size]

    def __str__(self):
        "Render as a string."
        return "Tree: Size %s" % self.size

class Lumberjack(object):
    "Represent a lumberjack who can cut down trees."
    def __init__(self):
        "Initialize: start with no tree."
        self.tree = None
        logging.info('Instantiated a Lumberjack')

    def cut_down_tree(self):
        "Convert tree to boards and go back to not having a tree."
        if not self.tree:
            raise TypeError("Cannot cut_down_tree(): Lumberjack has no tree!")
        boards = self.tree.get_boards()
        self.tree = None
        logging.info('Lumberjack.tree cut down')
        return boards

if __name__ == "__main__":
    "Demonstrate basic usage."
    john = Lumberjack()
    john.tree = Tree("XXL")
    if john.cut_down_tree() != 5:
        print("Error: XXL tree should yield 5 boards")
```

Clear the contents of the forestry.log file in your editor window, then save it as empty. Go ahead and run **test_forestry.py**, and then check **forestry.log** again (If you see a "Resource is out of sync" message, press **F5**). You'll see a nice list of log entries about progress made.

OBSERVE: Contents of log file after a run of test_forestry.py

```
INFO:root:Starting up the forestry program
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a Lumberjack
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
```

Also, *look* at the code. The logging messages are clearly logging messages. As you code, you'll find you mentally filter them out when you don't need them and they pop into focus when you do need them. This tends to be less obtrusive than print() calls that might be program-related or might be merely debugging information.

Other Logging Functions

The logging module makes it easy to flag issues with different levels of severity—in this next change, instead of **logging.debug()**, you'll use **logging.error()**. Try it out by adding **logging.error()** to the **__init__()** method of your Tree class and the **cut_down_tree()** method of the Lumberjack.

CODE TO EDIT: forestry.py

```
import logging

# set up the logger
~~logging.basicConfig(filename='forestry.log',level=logging.DEBUG)~~
logging.basicConfig(filename='forestry.log',level=logging.ERROR)

# log a message
logging.info('Starting up the forestry program')

class Tree(object):
    "Represent a tree in a forest that can be converted into boards."
    sizes = dict(S=1, M=2, L=3, XL=4, XXL=5)

    def __init__(self, size="L"):
        "Initialize: insist that size is a valid code."
        if size not in self.sizes:
            message = "Tree size must be one of: %s" % ",".join(self.sizes.keys())
            logging.error(message)
            raise ValueError(message)
        self.size = size
        logging.info('Instantiated a tree')

    def get_boards(self):
        "Return number of boards equivalent."
        logging.info('tree.get_boards method called')
        return self.sizes[self.size]

    def __str__(self):
        "Render as a string."
        return "Tree: Size %s" % self.size

class Lumberjack(object):
    "Represent a lumberjack who can cut down trees."
    def __init__(self):
        "Initialize: start with no tree."
        self.tree = None
        logging.info('Instantiated a Lumberjack')

    def cut_down_tree(self):
        "Convert tree to boards and go back to not having a tree."
        if not self.tree:
            ~~raise TypeError("Cannot cut_down_tree(): Lumberjack has no tree!")~~
            msg = "Cannot cut_down_tree(): Lumberjack has no tree!"
            logging.error(msg)
            raise TypeError(msg)
        boards = self.tree.get_boards()
        self.tree = None
        logging.info('Lumberjack.tree cut down')
        return boards

if __name__ == "__main__":
    "Demonstrate basic usage."
    john = Lumberjack()
    john.tree = Tree("XXL")
    if john.cut_down_tree() != 5:
        print("Error: XXL tree should yield 5 boards")
```

Now clear the forestry.log file again, and run **test_forestry.py**. Your tests should continue to pass:

OBSERVE: test_forestry.py results

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.008s
```

Now, check the new forestry.log:

added items to forestry.log

```
ERROR:root:Tree size must be one of: S,M,L,XL,XXL
ERROR:root:Cannot cut_down_tree(): Lumberjack has no tree!
```

With the Python logging library, you can set the logging level to filter out debug, info, warning, and error messages. The change we made at the beginning of the file ensured that only messages with ERROR or CRITICAL severity levels were even added to the log file.

The logging library includes these levels of built-in logger functions:

Level	Precedence	Description
DEBUG	10	Use for low-level debugging output
INFO	20	General information
WARNING	30	Warning messages such as deprecated functions and code
ERROR	40	Reporting exceptions and errors
CRITICAL	50	System crashes, security penetrations, data corruption, etc.

If the level at which you log a message is of lower priority than the level established for the logger when it is created, nothing is actually logged. This level of control is a good compromise, allowing you to easily suppress the logging of usually-unimportant messages without throwing away important ones.

Other Logging Levels

Logging presents a way to store data about programs in operation, and this is a good thing. But most of the time you do not want your program recording the mundane trivia of its existence. That is why you can specify a logging level when you create the logger. This will also log anything of higher precedence, so when you set it to ERROR, it also includes CRITICAL results. If you set the logging level to logging.INFO, it would show the INFO, WARNING, ERROR, and CRITICAL levels.

From now on, we'll set our logging level using a **start_logging()** function. Note that this uses a dict as a lookup table, allowing the caller to supply string values like "error" rather than having to import the numeric values from the logging module.

CODE TO EDIT: forestry.py

```
import logging
LOG_FILENAME = "forestry.log"
DEFAULT_LOG_LEVEL = "error" # Default log level
LEVELS = {'debug': logging.DEBUG,
          'info': logging.INFO,
          'warning': logging.WARNING,
          'error': logging.ERROR,
          'critical': logging.CRITICAL
         }

def start_logging(filename=LOG_FILENAME, level=DEFAULT_LOG_LEVEL):
    "Start logging with given filename and level."
    logging.basicConfig(filename=filename, level=LEVELS[level])
    # log a message
    logging.info('Starting up the forestry program')

~~# set up the logger~~
~~logging.basicConfig(filename='forestry.log',level=logging.ERROR)~~

~~# log a message~~
~~logging.info('Starting up the forestry program')~~

class Tree(object):
    "Represent a tree in a forest that can be converted into boards."
    sizes = dict(S=1, M=2, L=3, XL=4, XXL=5)

    def __init__(self, size="L"):
        "Initialize: insist that size is a valid code."
        if size not in self.sizes:
            message = "Tree size must be one of: %s" % ",".join(self.sizes.keys())
            logging.error(message)
            raise ValueError(message)
        self.size = size
        logging.info('Instantiated a tree')

    def get_boards(self):
        "Return number of boards equivalent."
        logging.info('tree.get_boards method called')
        return self.sizes[self.size]

    def __str__(self):
        "Render as a string."
        return "Tree: Size %s" % self.size

class Lumberjack(object):
    "Represent a lumberjack who can cut down trees."
    def __init__(self):
        "Initialize: start with no tree."
        self.tree = None
        logging.info('Instantiated a Lumberjack')

    def cut_down_tree(self):
        "Convert tree to boards and go back to not having a tree."
        if not self.tree:
            msg = "Cannot cut_down_tree(): Lumberjack has no tree!"
            logging.error(msg)
            raise TypeError(msg)
        boards = self.tree.get_boards()
        self.tree = None
        logging.info('Lumberjack.tree cut down')
        return boards

if __name__ == "__main__":
    "Demonstrate basic usage."
    john = Lumberjack()
    john.tree = Tree("XXL")
    if john.cut_down_tree() != 5:
        print("Error: XXL tree should yield 5 boards")
```

We now need to modify test_forestry.py to ensure that it still passes its tests. It needs to call the forestry module's start_logging function, which it does so with a level argument value of "error," which is automatically converted inside the function to logging.ERROR.

CODE TO EDIT: test_forestry.py

```
import unittest

from forestry import Lumberjack, Tree, start_logging

sizes = (("S", 1), ("M", 2), ("L", 3), ("XL", 4), ("XXL", 5))

class TestTree(unittest.TestCase):

    def test_lumber(self):
        for code, boards in sizes:
            tree = Tree(code)
            self.assertEqual(boards, tree.get_boards())

    def test_string(self):
        tree = Tree("L")
        self.assertEqual(str(tree), "Tree: Size L")

    def test_exceptions(self):
        self.assertRaises(ValueError, Tree, "parrot")
        self.assertRaises(TypeError, Lumberjack().cut_down_tree)

class TestLumberjack(unittest.TestCase):

    def test_lumberjack(self):
        for code, boards in sizes:
            tree = Tree(code)
            graham = Lumberjack()
            self.assertIsNone(graham.tree)
            graham.tree = tree
            brds = graham.cut_down_tree()
            self.assertIsNone(graham.tree)
            self.assertEqual(boards, brds)

if __name__ == "__main__":
    start_logging(level="error")
    unittest.main()
```

Getting Tests to Use Different Logging Levels

Right now, when you run test_forestry.py, it always runs under the ERROR level because it overrides the forestry.py default logging level. Which means all that is logged from the current code base is:

test_forestry.py results - Error level restricts output!

```
ERROR:root:Tree size must be one of: S,M,L,XL,XXL
ERROR:root:Cannot cut_down_tree(): Lumberjack has no tree!
```

Since you probably want as much information as possible to be generated by your unittests, you can do a local override of the logger configuration item by modifying the call to **start_logging** in test_forestry.py.

CODE TO EDIT: test_forestry.py

```
import unittest

from forestry import Lumberjack, Tree, start_logging

sizes = (("S", 1), ("M", 2), ("L", 3), ("XL", 4), ("XXL", 5))

class TestTree(unittest.TestCase):

    def test_lumber(self):
        for code, boards in sizes:
            tree = Tree(code)
            self.assertEqual(boards, tree.get_boards())

    def test_string(self):
        tree = Tree("L")
        self.assertEqual(str(tree), "Tree: Size L")

    def test_exceptions(self):
        self.assertRaises(ValueError, Tree, "parrot")
        self.assertRaises(TypeError, Lumberjack().cut_down_tree)

class TestLumberjack(unittest.TestCase):

    def test_lumberjack(self):
        for code, boards in sizes:
            tree = Tree(code)
            graham = Lumberjack()
            self.assertIsNone(graham.tree)
            graham.tree = tree
            brds = graham.cut_down_tree()
            self.assertIsNone(graham.tree)
            self.assertEqual(boards, brds)

if __name__ == "__main__":
    ~~start_logging(level="error")~~
    start_logging(level="info")
    unittest.main()
```

Run test_forestry.py to get these added log entries in forestry.log

```
INFO:root:Starting up the forestry program
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
INFO:root:Instantiated a tree
INFO:root:Instantiated a Lumberjack
INFO:root:tree.get_boards method called
INFO:root:Lumberjack.tree cut down
ERROR:root:Tree size must be one of: S,M,L,XL,XXL
INFO:root:Instantiated a Lumberjack
ERROR:root:Cannot cut_down_tree(): Lumberjack has no tree!
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
INFO:root:tree.get_boards method called
INFO:root:Instantiated a tree
```

Log Formatting

The log entries are providing a lot of information, but the default formatting we've used so far only provides a small subset of what the logger can capture for you. You'll use the log formatter to display significantly more data.

CODE TO EDIT: forestry.py

```
import logging
LOG_FILENAME = "forestry.log"
LOG_FORMAT = "%(asctime)s %(name)s:%(levelname)s:%(filename)s function:%(funcName)s line:%(lineno)d %(message)s"
DEFAULT_LOG_LEVEL = "warning" # Default log level
LEVELS = {'debug': logging.DEBUG,
          'info': logging.INFO,
          'warning': logging.WARNING,
          'error': logging.ERROR,
          'critical': logging.CRITICAL
         }

def start_logging(filename=LOG_FILENAME, level=DEFAULT_LOG_LEVEL):
    "Start logging with given filename and level."
    logging.basicConfig(filename=filename, level=LEVELS[level], format=LOG_FORMAT)
    # log a message
    logging.info('Starting up the forestry program')

class Tree(object):
    "Represent a tree in a forest that can be converted into boards."
    sizes = dict(S=1, M=2, L=3, XL=4, XXL=5)

    def __init__(self, size="L"):
        "Initialize: insist that size is a valid code."
        if size not in self.sizes:
            message = "Tree size must be one of: %s" % ",".join(self.sizes.keys())
            logging.error(message)
            raise ValueError(message)
        self.size = size
        logging.info('Instantiated a tree')

    def get_boards(self):
        "Return number of boards equivalent."
        logging.info('tree.get_boards method called')
        return self.sizes[self.size]

    def __str__(self):
        "Render as a string."
        return "Tree: Size %s" % self.size

class Lumberjack(object):
    "Represent a lumberjack who can cut down trees."
    def __init__(self):
        "Initialize: start with no tree."
        self.tree = None
        logging.info('Instantiated a Lumberjack')

    def cut_down_tree(self):
        "Convert tree to boards and go back to not having a tree."
        if not self.tree:
            msg = "Cannot cut_down_tree(): Lumberjack has no tree!"
            logging.error(msg)
            raise TypeError(msg)
        boards = self.tree.get_boards()
        self.tree = None
        logging.info('Lumberjack.tree cut down')
        return boards

if __name__ == "__main__":
    "Demonstrate basic usage."
    john = Lumberjack()
    john.tree = Tree("XXL")
    if john.cut_down_tree() != 5:
        print("Error: XXL tree should yield 5 boards")
```

This small change to the forestry framework makes a great deal of difference to the output in the logging stream. If you clear the log file and run **test_forestry.py**, your log should look like the following.

Results of test_forestry.py

```
2010-11-08 20:04:58,319 root:INFO:forestry.py function:start_logging line:17 Starting u
p the forestry program
2010-11-08 20:04:58,382 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,382 root:INFO:forestry.py function:__init__ line:46 Instantiated a
Lumberjack
2010-11-08 20:04:58,382 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,384 root:INFO:forestry.py function:cut_down_tree line:56 Lumberjack
.tree cut down
2010-11-08 20:04:58,384 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,384 root:INFO:forestry.py function:__init__ line:46 Instantiated a
Lumberjack
2010-11-08 20:04:58,384 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,384 root:INFO:forestry.py function:cut_down_tree line:56 Lumberjack
.tree cut down
2010-11-08 20:04:58,384 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,384 root:INFO:forestry.py function:__init__ line:46 Instantiated a
Lumberjack
2010-11-08 20:04:58,384 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,385 root:INFO:forestry.py function:cut_down_tree line:56 Lumberjack
.tree cut down
2010-11-08 20:04:58,385 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,385 root:INFO:forestry.py function:__init__ line:46 Instantiated a
Lumberjack
2010-11-08 20:04:58,385 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,387 root:INFO:forestry.py function:cut_down_tree line:56 Lumberjack
.tree cut down
2010-11-08 20:04:58,387 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,387 root:INFO:forestry.py function:__init__ line:46 Instantiated a
Lumberjack
2010-11-08 20:04:58,387 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,388 root:INFO:forestry.py function:cut_down_tree line:56 Lumberjack
.tree cut down
2010-11-08 20:04:58,388 root:ERROR:forestry.py function:__init__ line:27 Tree size must
 be one of: S,M,L,XL,XXL
2010-11-08 20:04:58,388 root:INFO:forestry.py function:__init__ line:46 Instantiated a
Lumberjack
2010-11-08 20:04:58,388 root:ERROR:forestry.py function:cut_down_tree line:52 Cannot cu
t_down_tree(): Lumberjack has no tree!
2010-11-08 20:04:58,388 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,388 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,388 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,390 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,391 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,391 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,391 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
2010-11-08 20:04:58,391 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,391 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
```

```
2010-11-08 20:04:58,391 root:INFO:forestry.py function:get_boards line:34 tree.get_boar
ds method called
2010-11-08 20:04:58,391 root:INFO:forestry.py function:__init__ line:30 Instantiated a
tree
```

You now have log entries that provide the date and time down to the microsecond for when the entry was recorded, the name of the file and the function/method that called it, and the line number if was called from. All of this from this line of Formatter String:

It should look like this but all on one line

```
%(asctime)s
    %(name)s:%(levelname)s:%(filename)s
        function:%(funcName)s line:%(lineno)d
    %(message)s
```

The **dark blue** elements above, such as "**function:**," are there to display the output in a more readable format. The **dark red** elements above are mapping keys that tell the logger where to put the data it collects. Some of the most useful keys are:

Key	Provides
%(name)	The owner of the log file
%(levelno)s	Numeric logging level for the message (DEBUG=10, INFO=20, WARNING=30, ERROR=40, CRITICAL=50)
%(levelname)s	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
%(pathname)s	Full pathname of the source file where the logging call was issued (if available)
%(filename)s	Filename portion of pathname
%(module)s	Module (name portion of filename)
%(funcName)s	Name of function/method containing the logging call
%(lineno)d	Source line number where the logging call was issued (if available)
%(asctime)s	Time when the log entry was created
%(message)s	The message passed into the log entry by the logger

It is often tempting to put *everything* into the log entry, but this can prove to be a mistake because too much text on a single line is hard for the human eye to interpret. In addition, if you have to scroll side-to-side on a log file you are prone to miss things. So here are some quick tips to making your log formats useful:

- Well-written messages make log files much more readable and searchable.
- Instead of adding **print()** calls, try changing your log format to include more information.
- %(pathname)s and %(filename)s are useful to identify the source of a message.
- Since you can search your code for log messages, recording the line number (%(lineno)d), though tempting, is less useful than you might imagine.
- Because the Python logging module can't capture which class objects generated an entry. the %(module)s and %(funcName)s keys can be troublesome.

The following is a reasonable example of a log file format:

OBSERVE: A Good Log File Format

```
%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

Logging isn't just a useful tool, it is like code comments and tests in that consistent use of it will impress experienced developers and good IT managers. That is because as much as good developers try to have all their code properly covered by tests, bugs creep in. Without logging, it can be nearly impossible to analyze sophisticated software behavior, uncover subtle errors, or see exactly step-by-step how a hacker tried to penetrate a system. With logging, you can provide a usage history that allows yourself and others to see what has happened with your programs. It is just another way to make processes visible. Don't be afraid of that visibility or the mistakes it may uncover; instead embrace it and learn from what is exposed.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Engineering Your Programs

In a previous lesson, we learned how to use **sys.argv** to access command line elements. However, **sys.argv** is only useful for providing per-run information about what you want a program to do. If there are actions you want a program to always take, you need a mechanism that allows that (while still allowing the use of command-line arguments for the per-run data).

For example, problems quickly arise if you need to accept several arguments. Let's say besides logging when you start a program, you need to point a program to a specific database, let the user set a specific directory to save files, and accept user name/password combinations. Now instead of one, you have five command-line arguments, each of which needs validation and precise help instructions. The logic to handle this would likely involve lots of nested **if** blocks to handle field determination/validation and **print()** calls for help instructions, and you'd spend significant effort, not just in writing and testing the command-line code, but also in maintaining it.

Thankfully, Python provides two libraries for handling this exact issue. The first library, **optparse**, is a more powerful command-line system than simply processing **sys.argv** "by hand." The second library, **configparser**, lets you create configuration files, often used to establish default program settings that can become, either for a single user or across a system, the defaults for command-line operation. This can sometimes shorten the "average" command line.

This lesson includes these sections:

- optparse: A Powerful Command-line Processor
- configparser: Controlling Settings the Right Way

optparse: A Powerful Command-line Processor

optparse is a convenient, flexible, and powerful library for parsing command-line options. It follows the conventional GNU/POSIX syntax, which sounds fancy but really just means that command-line users on Windows, Mac, Unix, Linux, and BSD will find it matches the general operation of their existing command-line tools.

A Simple optparse Example

Here we'll see how to capture a loglevel using the **optparse** library. This behavior may be useful to other programs, so we'll implement it in a new file. Create a **Python3_Lesson12** project, and assign it to the **Python3_Lessons** working set. In your **Python3_Lesson12** folder, create **commands.py** as shown:

CODE TO TYPE: commands.py

```
"""
commands.py: Parse logging level options from sys.argv
"""
from optparse import OptionParser

if __name__ == "__main__":

    # instantiate an OptionParser object
    parser = OptionParser()
    parser.add_option("-l", "--loglevel",
                        action="store",
                        dest="level",
                        default="warning",
                        help="set level of logger: debug, info, warning (default
), error, critical")
    (options, args) = parser.parse_args()
    print("level: %s" % options.level)
```

Now, let's try this out by using **-l debug** as command-line arguments.

Remember, to set up command-line arguments in Ellipse, first, select **Run | Run Configurations...** from the menu and click the left icon (**New Launch Configuration**) on the Run Configurations dialog toolbar. Enter **commands.py** in the Name field at the top of the dialog; for the Project, click **Browse** to select the **Python3_Lesson12** project; and for the Main Module, click **Browse** to select your **commands.py** program (in the src folder) as the program to run. Next, select the **Arguments** tab. In the Program Arguments field, enter **${string_prompt}**. This special value tells Ellipse to ask you for the arguments to the program when you run this configuration. Leave everything else as it is. Click **Run** at the bottom of the window. When prompted for the command-line arguments, enter **-l debug**.

commands.py called with '-l debug' argument

```
level: debug
```

Try the same thing with **-l critical** for a different result (you can just run **command.py** like you would run any Python program; Ellipse will remember to prompt you for the arguments):

commands.py called with '-l critical'

```
level: critical
```

Run it again, leaving the arguments field empty. It even provides a default value:

commands.py called without any argument

```
level: warning
```

Pretty handy, but besides a lot more typing, this isn't doing anything that **sys.argv** doesn't do, right? Lets go ahead and prove that assumption wrong. Run it with the **-h** argument:

commands.py called with -h argument

```
Usage: commands.py [options]

Options:
  -h, --help            show this help message and exit
  -l LEVEL, --loglevel=LEVEL
                        set level of logger: debug, info, warning (default),
                        error, critical
```

There you have it—instant help! And help that follows the same format that you get any time you do '-h' or '--h' on a command-line tool. Also, note that the **print()** command did not run. This is because all Python programs, regardless of whether or not they use the **optparse** library, do not run any code except to produce the help text when the user calls for help. So users can call the **-h** command without fear that they will inadvertently run your program.

A Complex optparse Example

Let's do something familiar and create a very simple email address book program. It will allow you to add, delete, and list all addresses from the command line. The addresses will be stored using the shelve module.

The first thing to do is to get our program to add and delete emails. You'll need two options for this, the first one to let your users pick the add, edit, or delete actions, and the second being the email value in question. So create **addressbook.py** as shown below. You will need to add another run configuration that allows you to set the command-line arguments when you run it, just like you did for commands.py.

CODE TO TYPE: addressbook.py

```
from optparse import OptionParser

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option('-a', '--action', dest="action", action="store", help="req
uires -e option. Actions: add/delete")
    parser.add_option('-e', '--email', dest="email", action="store", help="email
 used in the -a option")
    (options, args) = parser.parse_args()
```

Save and run it with the **-h** option:

OBSERVE: Running addressbook.py With -h Argument

```
Usage:
addressbook.py [options]
Options: -h, --help show this help message and exit
         -a ACTION, --action=ACTION requires -e option.
                    Actions: add/edit/delete
         -e EMAIL, --email=EMAIL email used in the -a option
```

Validating optparse Options

Now, we'll add some more validation. First we'll check that, when a user provides the **--action** option, they also provide an **--email** option. Then we'll check that the email provided is valid (for the sake of simplicity, we'll just check that it contains the "@" character).

The first validation, that --action has an --email (and vice versa) is done by checking that if one of those options exists, so should the other. If only one exists, a **parser.error()** is called. Edit **addressbook.py** as shown:

CODE TO EDIT: addressbook.py

```
from optparse import OptionParser

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option('-a', '--action', dest="action", action="store",
                            help="requires -e option. Actions: add/delete")
    parser.add_option('-e', '--email', dest="email", action="store",
                            help="email used in the -a option")
    (options, args) = parser.parse_args()
    # validation
    if options.action and not options.email:
        parser.error("option -a requires option -e")
    elif options.email and not options.action:
        parser.error("option -e requires option -a")
    print(options)
```

Save and run it with **-a add**:

OBSERVE: Running addressbook.py with -a add

```
Usage: addressbook.py [options]

addressbook.py: error: option -a requires option -e
```

Now, run it with **-e steve@oreilly.com**:

OBSERVE: Running addressbook.py with -e steve@oreilly.com

```
Usage: addressbook.py [options]

addressbook.py: error: option -e requires option -a
```

The requirement for both options to appear together is working, so it only remains to ensure that when both options are present they are correctly captured in the **options** dict. You can do this by running the program with both options.

Run it with **-a steve -e something**:

OBSERVE: Running addressbook.py with -a steve -e something

```
{'action': 'steve', 'email': 'something'}
```

Now, can you figure out how to validate that user-provided email includes "@"? Try it before we show you!

...

...

...

...

...

Go ahead; *try* it!

...

...

...

...

...

We're waiting!

...

...

...

...

...

You should have arrived at something like this:

CODE TO EDIT: addressbook.py

```
from optparse import OptionParser

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option('-a', '--action', dest="action",
            action="store", help="requires -e option. Actions: add/delete")
    parser.add_option('-e', '--email', dest="email",
            action="store", help="email used in the -a option")
    (options, args) = parser.parse_args()

    # validation
    if options.action and not options.email:
        parser.error("option -a requires option -e")
    elif options.email and not options.action:
        parser.error("option -e requires option -a")
    elif options.email and '@' not in options.email:
        parser.error("option -e requires a valid email address")
    print(options)
```

Run it with **-a steve -e something**:

OBSERVE: Running addressbook.py with -a steve -e something

```
Usage: addressbook.py [options]

addressbook.py: error: option -e requires a valid email address
```

Showtime!

Okay, it's time to add the code that handles the emails. But before we do that, let's add the obligatory tests. Create **test_addressbook.py** as shown:

CODE TO TYPE: test_addressbook.py

```
import unittest
import addressbook

class TestEmailHandlers(unittest.TestCase):

    def setUp(self):
        self.email = 'test123@t.com'

    def test_email_delete(self):
        addressbook.email_add(self.email) # ensure the email is active
        self.assertEqual(addressbook.email_delete(self.email)[0], True)
        self.assertEqual(addressbook.email_delete(self.email)[0], False)

    def test_email_add(self):
        self.assertEqual(addressbook.email_add(self.email)[0], True)
        self.assertEqual(addressbook.email_add(self.email)[0], False)

if __name__ == "__main__":
    unittest.main()
```

Now edit the **addressbook** program to accommodate the tests:

CODE TO EDIT: addressbook.py

```
from optparse import OptionParser
import shelve
import sys

shelf_location = 'V:/workspace/Python3_Lesson12/src/email.shelf'

def email_add(email):
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    if email in emails:
        message = False, 'Email "%s" already in address book' % email
    else:
        emails.append(email)
        message = True, 'Email "%s" added to address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def email_delete(email):
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    try:
        emails.remove(email)
        message = True, 'Email "%s" removed from address book' % email
    except ValueError:
        message = False, 'Email "%s" was not in the address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def main(options):
    "routes requests"
    if options.action == 'add':
        return email_add(options.email)
    elif options.action == 'delete':
        return email_delete(options.email)

if __name__ == '__main__':
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    shelf.close()
    parser = OptionParser()
    parser.add_option('-a', '--action', dest="action", action="store",
                            help="requires -e option. Actions: add/delete")
    parser.add_option('-e', '--email', dest="email",
                            action="store", help="email used in the -a option")

    (options, args) = parser.parse_args()
    # validation
    if options.action is None:
        sys.exit("You must specify an action (add or delete) with '-a action'")
    if options.action and not options.email:
        parser.error("option -a requires option -e")
    elif options.email and not options.action:
        parser.error("option -e requires option -a")
    elif options.email and '@' not in options.email:
        parser.error("option -e requires a valid email address")
    ~~print(options)~~
    print(main(options)[1])
```

Of course, you wrote tests for this code before writing it. Better try out the tests before trying to exercise the code. Save both programs and run **test_addressbook.py**:

OBSERVE: Testing with test_addressbook.py

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Well, that seemed to work out OK, or at least the tests seem to indicate that the add and delete functionality is succeeding and failing where expected. So let's see what we get with various calls from the command line.

Run addrbook.py with **-a add -e steve@h.com**:

OBSERVE: Running addressbook.py with -a add -e steve@h.com

```
Email "steve@h.com" added to address book
```

Run addrbook.py with **-a add -e steve@h.com** again:

OBSERVE: Running addressbook.py again with -a add -e steve@h.com

```
Email "steve@h.com" already in address book
```

Run addrbook.py with **-a delete -e steve@h.com**:

OBSERVE: Running addressbook.py with -a delete -e steve@h.com

```
Email "steve@h.com" removed from address book
```

Run addrbook.py with **-a delete -e steve@h.com** again:

OBSERVE: Running addressbook.py again with -a delete -e steve@h.com

```
Email "steve@h.com" was not in the address book
```

You've now got a grip on quite a few of the fundamentals of using **optparse**. Notice how, once the code gets past **optparse** validation, the action turns to functions. This makes things much easier to extend and test, in turn helping you to reuse this code in other modules. In fact, the email validation ought to take place in its own function called by the email handlers, and would raise an exception that would be caught by the parse handler. Something like this could work:

CODE TO EDIT: addressbook.py

```
from optparse import OptionParser
import shelve
import sys

shelf_location = 'V:/workspace/Python3_Lesson12/src/email.shelf'

class InvalidEmail(Exception):
    pass

def validate_email(email):
    if '@' not in email:
        raise InvalidEmail("Invalid email: "+email)

def email_add(email):
    validate_email(email)
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    if email in emails:
        message = False, 'Email "%s" already in address book' % email
    else:
        emails.append(email)
        message = True, 'Email "%s" added to address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def email_delete(email):
    validate_email(email)
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    try:
        emails.remove(email)
        message = True, 'Email "%s" removed from address book' % email
    except ValueError:
        message = False, 'Email "%s" was not in the address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def main(options):
    "routes requests"
    if options.action == 'add':
        return email_add(options.email)
    elif options.action == 'delete':
        return email_delete(options.email)

if __name__ == '__main__':
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    shelf.close()
    parser = OptionParser()
    parser.add_option('-a', '--action', dest="action", action="store",
                        help="requires -e option. Actions: add/delete")
    parser.add_option('-e', '--email', dest="email",
                        action="store", help="email used in the -a option")

    (options, args) = parser.parse_args()
    # validation
    if options.action is None:
        sys.exit("You must specify an action (add or delete) with '-a action'")
    if options.action and not options.email:
        parser.error("option -a requires option -e")
    elif options.email and not options.action:
        parser.error("option -e requires option -a")
    ~~elif options.email and '@' not in options.email:~~
        ~~        parser.error("option -e requires a valid email address")~~
    ~~    print(main(options)[1])~~
    try:
        print(main(options)[1])
    except InvalidEmail:
        parser.error("option -e requires a valid email address")
```

Go ahead and test this code. All tests should continue to pass.

OBSERVE: Refactored code passes all tests

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.016s

OK
```

Then run the program itself. It should work just as before, but the refactoring makes the code more easily extended.

Displaying All the Records

Now, suppose we want to list the contents of the shelf file. For this, all we need is an option without a value. Perhaps just **-d** or **--display** without a value to show every address in the system. We'll do it with this parser option:

boolean flag parser option

```
 parser.add_option('-d',
        '--display', dest="display", action="store_true", help="show all emails"
)
```

In this parser option, the action of 'store_true' means that if you call it via the box above, the display attribute of options will be a boolean True. Otherwise it is a **None** object. With that in your tool-chest, you can add to your existing code base. First, as usual, we'll add a test for the new functionality. Edit **test_addressbook.py** as shown:

CODE TO EDIT: test_addressbook.py

```
import unittest, shelve
import addressbook

class TestEmailHandlers(unittest.TestCase):

    def setUp(self):
        self.email = 'test123@t.com'
        shelf_location = addressbook.shelf_location

        shelf = shelve.open(shelf_location)
        if 'emails' in shelf:
            if self.email in shelf['emails']:
                shelf['emails']=[]
        shelf.close()

    def test_email_delete(self):
        addressbook.email_add(self.email) # ensure the email is active
        self.assertEqual(addressbook.email_delete(self.email)[0], True)
        self.assertEqual(addressbook.email_delete(self.email)[0], False)

    def test_email_add(self):
        self.assertEqual(addressbook.email_add(self.email)[0], True)
        self.assertEqual(addressbook.email_add(self.email)[0], False)

    def test_email_display(self):
        addressbook.email_add(self.email)
        val, display = addressbook.email_display()
        self.assertTrue(self.email in display)

if __name__ == "__main__":
    unittest.main()
```

Now, add the functionality; edit **addressbook.py** as shown:

CODE TO EDIT: addressbook.py

```
from optparse import OptionParser
import shelve
~~import sys~~

shelf_location = 'V:/workspace/Python3_Lesson12/src/email.shelf'

class InvalidEmail(Exception):
    pass

def validate_email(email):
    if '@' not in email:
        raise InvalidEmail("Invalid email: "+email)

def email_add(email):
    validate_email(email)
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    if email in emails:
        message = False, 'Email "%s" already in address book' % email
    else:
        emails.append(email)
        message = True, 'Email "%s" added to address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def email_delete(email):
    validate_email(email)
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    try:
        emails.remove(email)
        message = True, 'Email "%s" removed from address book' % email
    except ValueError:
        message = False, 'Email "%s" was not in the address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def email_display():
    shelf = shelve.open(shelf_location)
    emails = shelf['emails']
    shelf.close()
    text = ''
    for email in emails:
        text += email + '\n'
    return True,text

def main(options):
    "routes requests"
    if options.action == 'add':
        return email_add(options.email)
    elif options.action == 'delete':
        return email_delete(options.email)
    elif options.display == True:
        return email_display()

if __name__ == '__main__':
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    shelf.close()
    parser = OptionParser()
    parser.add_option('-a', '--action', dest="action", action="store",
                            help="requires -e option. Actions: add/delete")
    parser.add_option('-e', '--email', dest="email",
                            action="store", help="email used in the -a option")

    parser.add_option('-d', '--display', dest="display", action="store_true",
                            help="show all emails")
    (options, args) = parser.parse_args()
    # validation
    ~~if options.action is None:~~
        ~~sys.exit("You must specify an action (add or delete) with '-a action'")~~
    if options.action and not options.email:
        parser.error("option -a requires option -e")
    elif options.email and not options.action:
        parser.error("option -e requires option -a")
    try:
        print(main(options)[1])
    except InvalidEmail:
        parser.error("option -e requires a valid email address")
```

Run your tests, and then add some emails and run **addressbook.py** with the **-d**/--display flag. You'll get a printed display of all your email entries.

optparse Type Validation

Let's say we want to change the -d/--display flag to provide a number of records based on an integer we pass in. Normally that means we'll have to do type checking via the **int()** built-in, but with **optparse**, we get a shortcut.

OBSERVE: Adding an Integer Check

```
parser.add_option('-d', '--display', dest="display", type="int",
    action="store~~_true~~", help="show all emails limited by value")
```

The **optparse** module also includes type checking for string, float, and *choices*. These should all be obvious, except for *choices*.

The **optparse** module lets you easily write scripts that handle arguments in a fashion that is consistent with the rest of the world. In other words, it is common to put all of your optparse code under the **if __name__ == '__main__'** block of code, since that means if another module extends your code it doesn't trigger the optparse code in your program.

configparser: Controlling Settings the Right Way

Let's say you just bought a brand new computer. The first time you start it up, the computer asks you your name, password, time zone, language, and probably some other questions. It isn't hard to do, but it takes away from your time with your new machine. Wouldn't it be nice if you could simply save this configuration information on one computer and place it on another as needed?

Actually, you can. System Engineers often use tools that set up computers with all the configuration information set exactly how they want it. With some automated scripting, they can start up a new computer this way in minutes and sometimes seconds. This is how companies that provide hosting for individuals or firms that run gigantic server farms can maintain hundreds and thousands of machines.

Python's **configparser** library provides an easily used API for interacting with one of the popular formats used to save configurations, the INI file format. Frequently associated with Microsoft Windows, INI is in fact also used by other platforms such as Linux and Mac OS X.

configparser to Store Database Settings

In previous courses and earlier in this lesson, we used simple files, pickle, shelve, or SQL databases to save information. The information that handled your settings was coded right into your programs. While this works on small projects under well-defined academic conditions such as Eclipse, Ellipse, and the O'Reilly teaching environment, it can be problematic under professional conditions. For example, because Python is so portable you might save data on Windows at **c:\data\emails.shelf**, but this simply won't work on Linux or

Mac OS X, which might want to see something like **/usr/local/data/emails.shelf**. Python has tools that make it easy to detect operating systems, but then users might want to save their data in a specific location. This forces them to change your code to store data where they want, which introduces the risk of breaking your code, and only works if they were actually given access to your code (source files).

This is where config files can be priceless. Users not familiar with Python can quickly figure out the format and change things. Furthermore, since the file usually has a **.cfg** (or less commonly, **.ini**) extension, most users will be able to quickly identify it as a configuration file.

So, let's make a configuration file. Create **addressbook.cfg** as shown:

CODE TO TYPE: addressbook.cfg

```
[database]
# mac os x or linux
# file = /workspace/Python3_lesson12/src/email.shelf
# windows
file = V:\workspace\Python3_lesson12\src\email.shelf
```

[database] is a section header. That means any option variables defined under it use "database" as part of the process of displaying them. Under that are a series of comments that use Python '#' syntax so that they are not loaded. Finally, **file = V:\workspace\Python3_lesson12\src\email.shelf** sets the file variable under the database section. To display this addressbook.cfg file, create a **config.py** file as shown:

CODE TO TYPE: config.py

```
import configparser

# create a config parser object
config = configparser.RawConfigParser()

# open and read the addressbook.cfg file into the config parser
config.read('addressbook.cfg')

# loop through the sections
for section in config.sections():
    print(section)
    # get all the options for the current section
    for option in config.options(section):
        # print the option and its value indented for clarity
        text = '    %s = %s' % (option, config.get(section, option))
        print(text)
```

Save and run it:

OBSERVE: the results of running config.py

```
database
    file = V:\workspace\Python3_lesson12\src\email.shelf
```

As you can see, this gives us the ability to provide per-system config files. This is a good thing, because it means you don't have to worry so much about users needing to change settings. A system administrator can establish a central configuration file (and savvy users can provide their own configurations). Let's use the addressbook.cfg file to set the database location in addressbook.py:

CODE TO EDIT: addressbook.py

```
import configparser
from optparse import OptionParser
import shelve

~~shelf_location = 'V:/workspace/Python3_Lesson12/src/email.shelf'~~
config = configparser.RawConfigParser()
config.read('V:/workspace/Python3_Lesson12/src/addressbook.cfg')
shelf_location = config.get('database', 'file')

class InvalidEmail(Exception):
    pass

def validate_email(email):
    if '@' not in email:
        raise InvalidEmail("Invalid email: "+email)

def email_add(email):
    validate_email(email)
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    if email in emails:
        message = False, 'Email "%s" already in address book' % email
    else:
        emails.append(email)
        message = True, 'Email "%s" added to address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def email_delete(email):
    validate_email(email)
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    emails = shelf['emails']
    try:
        emails.remove(email)
        message = True, 'Email "%s" removed from address book' % email
    except ValueError:
        message = False, 'Email "%s" was not in the address book' % email
    shelf['emails'] = emails
    shelf.close()
    return message

def email_display():
    shelf = shelve.open(shelf_location)
    emails = shelf['emails']
    shelf.close()
    text = ''
    for email in emails:
        text += email + '\n'
    return True,text

def main(options):
    "routes requests"
    if options.action == 'add':
        return email_add(options.email)
    elif options.action == 'delete':
        return email_delete(options.email)
    elif options.display == True:
        return email_display()

if __name__ == '__main__':
    shelf = shelve.open(shelf_location)
    if 'emails' not in shelf:
        shelf['emails'] = []
    shelf.close()
    parser = OptionParser()
    parser.add_option('-a', '--action', dest="action", action="store",
                            help="requires -e option. Actions: add/delete")
    parser.add_option('-e', '--email', dest="email",
                            action="store", help="email used in the -a option")

    parser.add_option('-d', '--display', dest="display", action="store_true",
                            help="show all emails")
    (options, args) = parser.parse_args()
    # validation
    if options.action and not options.email:
        parser.error("option -a requires option -e")
    elif options.email and not options.action:
        parser.error("option -e requires option -a")
    try:
        print(main(options)[1])
    except InvalidEmail:
        parser.error("option -e requires a valid email address")
```

Save and run your tests and your code. There should be no difference in the results. Now, let's see what happens when we don't provide a file option. Comment it out in the cfg file as shown:

CODE TO EDIT: addressbook.cfg

```
[database]
# mac os x or linux
# file = /workspace/Python3_lesson12/src/email.shelf
# windows
#file = V:\workspace\Python3_lesson12\src\email.shelf
```

Save it and run your **addressbook.py**.

OBSERVE: Running addressbook.py with no defined database.

```
Traceback (most recent call last):
    File "addressbook.py", line 7, in <module>
        shelf_location = config.get('database', 'file')
    File "configparser.py", line 327, in get
        raise NoOptionError(option, section)
configparser.NoOptionError: No option 'file' in section: 'database'
```

Your code has just thrown an exception! You can have the **config.get()** method pass in a default, but usually you want to leave these exceptions as they are. Users who play with config files need as much information as possible to get their configurations working, and passing in defaults means they may not understand what happens when they pass in a setting incorrectly. Remember, good Python programmers like to be as explicit as possible!

Multiple Sections

Here's a sample config file that might be used to set up a computer. This is just a simple example to show you how system engineers often work:

Operating System Basic Setup

```
[personal]
first_name = Steve
last_name = Holden
age = 33
gender = male

[professional]
occupation = author
website = http://holdenweb.com

[location]
language = English (USA)
timezone = EST

[authentication]
username = sholden
password = like I'm going to tell you
```

The **configparser** tool is extremely readable and quite machine-friendly. A significant portion of the Python community uses the INI format to describe critical dependency lists and so do users from other programming languages. While there are other competing formats, such as XML, the INI format remains popular because quite simply it is easy for humans to read and for machines it is as fast as, if not faster than, the others to parse and interpret. This ease of interpretation has meant that XML usage for configuration has declined in recent years while use of the older INI format has grown.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).

Time-Based Computations

What time is it?

If it takes seven days for the check to arrive, on what day will it arrive?

How many days until my birthday?

While these might seem amusing, they are serious—and common—questions for software developers. Time-based functions are invariably complex because we track time by non-decimal methods. While we might have 10 fingers and a meter has 100 centimeters, when it comes to time an hour has 60 minutes, a week has 7 days, and a month can have from 28 to 31 days. Also, almost every four years you have to account for leap year. Many day-tracking calculations have to take into account the standard business days of Monday through Friday, and the weekend days of Saturday and Sunday. The list of "edge cases" in time calculations is almost infinite!

It is arguably for this reason that Python has three built-in libraries for handling time issues: **datetime**, **time**, and **calendar**, each of which has a lot of sophisticated functionality. Because of the volume of functionality provided by each library, this lesson will focus on the **datetime** library. In fact, this lesson will focus on the three questions asked at the start, since they provide an excellent introduction to many features of handling time from the perspective of a software developer.

This lesson includes these sections:

- What Time is It?
- If It Takes Thirty-One Days...?
- How Many Days Until My Birthday?

What Time is It?

A common way to find the current time is with the code in this interactive session:

CODE TO TYPE: Type this code in an interactive console session

```
>>> import datetime
>>> print(datetime.datetime.now())
2010-09-26 20:21:50.813824
```

And there you have the time!

Time Representations

How a date is formatted depends on who is looking at it. For a software developer, engineer, system administrator, or scientist, the format shown in that last session is a good way to see time. Because the date is in YYYY-MM-DD format and the time uses the 24-hour clock, you can do easy sorting on the results either by hand or with computers, whereas the American (MM/DD/YYYY) and European (DD/MM/YYYY) date methods require more work for sorting, and the 12-hour clock repeats itself, so times after noon won't sort correctly.

In fact, often people working in these time-sensitive fields rely on alternate time measurement methods like counting seconds since the epoch or the Julian date (JD) system used by the astronomy community. Python supports these alternate methods extremely well, which is one minor reason why Python is so frequently used by the scientific community.

However, most people don't like, or even understand, this way of representing time. It isn't what they're used to seeing and forcing them to use a new time representation format is a good way to lose their interest in your projects. Let's do some formatting to make this a little more natural to the American eye. Continue your interactive session:

CODE TO TYPE: Type this code in an interactive console session

```
>>> now = datetime.datetime.now()
>>> format_string = "%x %X"
>>> now.strftime(format_string)
09/26/10 20:35:04
```

From previous lessons, you know what a formatter string does. Now nearly every time object by Python supports the **strftime()** method, which accepts a format string with any number of predefined mapping keys. **"%x %X"** fetches the datetime setup you defined on your computer when you set it up.

These predefined mapping keys let you actually precisely map exactly what date and time setup you want your users to experience. The legal mapping keys are:

key	**Meaning**
%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].
%f	Microsecond as a decimal number [0,999999], zero-padded on the left
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale's equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%z	UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).
%Z	Time zone name (empty string if the object is naive).
%%	A literal '%' character.

Armed with this table and the strftime() function, you can now provide a much more attractive date format customized for your target user. Continue your interactive session now to try it out:

CODE TO TYPE: Type this code in an interactive console session

```
>>> format_string = "%A, %B %d, %Y at %I:%M %p."
>>> now.strftime(format_string)
Sunday, September 26, 2010 at 8:35 PM.
```

If it Takes Thirty-One Days...?

At a glance this should be easy—you just take the date of the month as fetched by **datetime.datetime.now()** and add 31, right? Lets give it a try. Create a **Python3_Lesson13** project and assign it to the **Python3_Lessons** working set. Then, in your **Python3_Lesson13/src** folder, create **count_thirtyone_days.py** as shown:

CODE TO TYPE: count_thirtyone_days.py

```
import datetime
now = datetime.datetime.now()
date = now.strftime("%d")
delivery = int(date) + 31
print("Today: %s" % date)
print("Delivery: %s" % delivery)
```

At a glance, this looks like it should work, but in fact you'll get a response like this:

OBSERVE: Running count_thirtyone_days.py on November 29th

```
Today: 29
Delivery: 60
```

In theory, you could write a bit of code that would handle month rollovers and the leap year. This is not a small undertaking and will probably take more time than you really want to dedicate to the problem of adding thirty-one days to the current date. Also, your result would lack the ability to reformat the results via the **strftime()** method because it would be a simple integer, not a time object.

There Must be a Better Way to Add Days to a Date!

Yes, there is a way. The Python datetime library has an object named **timedelta**, which represents the difference between two dates or times. This difference is called a *duration*. You can add a timedelta to the current date, and it will account for month rollovers and the leap year. Modify **count_thirtyone_days.py** as shown:

CODE TO EDIT: count_thirtyone_days.py

```
from datetime import datetime, timedelta # more attractive import
now = ~~datetime.~~datetime.now()
delta = timedelta(31) # create a timedelta of 31 days
delivery = now + delta # add the timedelta to the current datetime.
print("Today: %s" % now.strftime("%d"))
print("Delivery: %s" % delivery.strftime("%d"))
```

Save and run it. You'll see that it works correctly:

Running count_thirtyone_days.py on November 29th

```
Today: 29
Delivery: 30
```

You may have noticed that what was printed was string values returned from the **strftime()** methods on the **now()** and **delivery** objects. This means that you can execute further calculations as needed on these objects—they have not been changed at all. This becomes really useful when you want to skip over weekends. Thanks to the **datetime** object's **isoweekday()** method which returns a numeric value as shown below, we can write code that skips over weekends with some ease.

Value returned from isoweekday()	Weekday name
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

The next example shows how to skip over weekends. It doesn't take into account national or bank holidays, but it is similar to what organizations use to determine when they can expect payments and other letters to arrive.

CODE TO TYPE: Enter the code below as skip_weekdays.py

```
from datetime import datetime, timedelta

delivery = datetime.now()
delta = timedelta(1)
count = 0
while count < 31:
    delivery = delivery + delta
    if delivery.isoweekday() in (6, 7):
        continue
    count += 1

now = datetime.now()
print(now)
print(delivery)
print("Today: %s" % now.strftime("%d"))
print("Delivery: %s" % delivery.strftime("%d"))
```

Save and run it. It counts only working days.

OBSERVE: Running skip_weekdays.py on November 29th

```
2010-11-29 10:40:26.439000
2011-01-11 10:40:26.439000
Today: 29
Delivery: 11
```

timedeltas for Weeks, Hours, Minutes, and Seconds

The **timedelta** object can be instantiated with other values than days. Some of the ones you'll use frequently are weeks, hours, minutes, and seconds. All you need to do is add one or more of these items as arguments and the timedelta is constructed accordingly. Create **more_deltas.py** as shown to see what you get:

CODE TO TYPE: more_deltas.py

```
from datetime import datetime, timedelta

weeks = timedelta(weeks=2)
hours = timedelta(hours=1)
minutes = timedelta(minutes=100)
seconds = timedelta(seconds=1000)
composite = timedelta(hours=1, minutes=30)

now = datetime.now()
print(now)
print(now + weeks)
print(now + hours)
print(now + minutes)
print(now + seconds)
print(now + composite)
```

Save and run it (your results will vary unless you traveled back in time to November 29, 2010):

OBSERVE: Running more_deltas.py

```
2010-11-29 10:41:06.312000
2010-12-13 10:41:06.312000
2010-11-29 11:41:06.312000
2010-11-29 12:21:06.312000
2010-11-29 10:57:46.312000
2010-11-29 12:11:06.312000
```

timedeltas for Years and Months

Years and months are not constants, thanks to the leap year issue and the general inconsistency of month durations. Therefore, the timedelta does not accept them as arguments. However, because the other arguments (weeks, hours, minutes, etc.) are constants, timedeltas can handle the leapyear and month durations, which works well for years and not so well for months.

This means you can provide an almost exact year timedelta by simply doing this:

CODE TO TYPE: Type this code in an interactive console session

```
>>> from datetime import timedelta
>>> timedelta(365)
datetime.timedelta(365)
```

On the other hand, this obviously fails because months range in duration from 28 to 31 days:

CODE TO TYPE: Type this code in an interactive console session

```
>>> timedelta(30)
datetime.timedelta(30)
```

The general indeterminate duration of a month is exactly why bankers use 30 days as their standard value and why scientists prefer other date formats.

How Many Days Until my Birthday?

Remember when you were a kid and carefully counted the days until your next birthday? As a programmer you can skip marking off each day and simply write a program to do the work for you. You can write a simple program that:

1. Takes your birthday

2. Converts your birthday to a datetime object
3. Subtracts the current date from your birthday object
4. Publishes the results

Ready? Let's do this thing!

When is Your Birthday?

The first step is to accept a date as your birthday. Let's use optparse to accept a string to be converted into a datetime object. Then we'll use a new method, **datetime.strptime()** to not only convert the string to a date, but confirm that it is a valid date. The **datetime.strptime()** method works like **datetime.strftime()**, but in reverse, converting strings to date objects. You use the same date-formatting keys as described for **datetime.strftime()**, which means you can create common formatting strings used across your application for both creating and rendering time objects.

CODE TO TYPE: Type this code in an interactive console session

```
>>> formatter_string = "%m-%d-%Y" # format for MM-DD-YYYY
>>> from datetime import datetime
>>> datetime.strptime("07-24-1967", formatter_string) # The conversion code
datetime.datetime(1967, 7, 24, 0, 0)
```

But what if someone enters a date such as "1967-07-24" or something like "Python ROCKS" or even "15-35-2010"? Since those does not match the format specified by the formatter string and are not valid dates, **datetime.strptime()** throws a ValueError exception. This makes it trivial to create datetime validators without having to lean on string methods or even regular expressions, which could handle the rough formatting issue of numbers, but can't as easily handle the confirmation that a date is real.

With what we've learned so far, let's write some **birthday.py** code:

CODE TO TYPE: birthday.py

```
import logging
from datetime import datetime
from optparse import OptionParser

logging.basicConfig(filename='birthday.log',level=logging.DEBUG)

class InvalidDateFormat(Exception):
    pass

def string_to_date(date):
    """
    Converts 'MM-DD-YYYY' to a date/time object
        or throws an InvalidDateFormat exception
    """
    try:
        # create a datetime object from the date value
        formatter_string = "%m-%d-%Y"
        birthday = datetime.strptime(date, formatter_string)
    except ValueError as e:
        # log the format error then raise it again so it can be handled graceful
ly
        logging.error(e)
        raise InvalidDateFormat(e)
    return birthday

def birthday_counter(birthday):
    """
    Returns the number of days until your birthday.
        (not yet fully implemented)
    """
    return 100

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option('-b', '--birthday', dest="birthday", action="store",
    help="Your birthday in MM-DD-YYYY format")
    (options, args) = parser.parse_args()

    format_error_message = "birthday.py requires a date in MM-DD-YYYY format"
    if not options.birthday:
        parser.error(format_error_message)

    try:
        print(birthday_counter(options.birthday))
    except InvalidDateFormat:
        parser.error(format_error_message)
```

This looks pretty good, but how do you know it works? Time to write a unittest!

CODE TO TYPE: test_birthday.py

```
from datetime import datetime
import unittest


from birthday import *

class TestBirthday(unittest.TestCase):

    def test_birthday_counter(self):
        self.assertEqual(birthday_counter("10-31-1948"), 100)

    def test_string_to_date(self):

        self.assertRaises(InvalidDateFormat, string_to_date, "10-32-1948")
        # create a new datetime object from scratch
        datetime_obj = datetime(2012, 10, 31)
        self.assertEqual(datetime_obj, string_to_date("10-31-2012"))

if __name__ == "__main__":
    unittest.main()
```

Save and run it; both tests pass. Take a careful look at the second test, which checks that the **string_to_date()** function works properly. To do that, its second assertion requires a datetime created from scratch. Hence this line of code:

CODE TO TYPE: Type this code in an interactive console session

```
>>> from datetime import datetime
>>> datetime(2012, 10, 31)
datetime.datetime(2010, 10, 31, 0, 0)
```

Note that the **self.assertEqual(datetime_obj, string_to_date("10-31-1948"))** assertion is actually just doing **datetime_obj == string_to_date("10-31-1948")**. Just as you can add or subtract datetime objects to or from each other, you can also do comparisons against them. This means you can do any of these comparisons:

Sign	Description
==	equals
>	greater than
>=	greater than or equals
<	less than
<=	less than or equals

More Ways to Construct Dates

You can get a lot more specific than days. You can specify hours, minutes, seconds, and microseconds. This is good for constructing tests and setting up deadlines and other time-related points. Create **making_time.py** as shown:

CODE TO TYPE: making_time.py

```
from datetime import datetime
print(datetime(2012, 10, 31))
print(datetime(2012, 10, 31, 12))
print(datetime(2012, 10, 31, 12, 30))
print(datetime(2012, 10, 31, 12, 30, 59))
print(datetime(2012, 10, 31, 12, 30, 59, 300))
```

Save and run it:

OBSERVE: Results from Running making_time.py

```
2012-10-31 00:00:00
2012-10-31 12:00:00
2012-10-31 12:30:00
2012-10-31 12:30:59
2012-10-31 12:30:59.000300
```

Fetching Years, Months, Hours, etc. from a Datetime Object

The datetime object has integer attributes that are specific year, month, day, hour, minute, second, and microsecond representations for that object. Create **time_attributes.py** as shown below to demonstrate your options:

code to enter: time_attributes.py

```
from datetime import datetime
dt = datetime(2012, 10, 31, 12, 30, 59, 300)
print(dt.year)
print(dt.month)
print(dt.day)
print(dt.hour)
print(dt.minute)
print(dt.second)
print(dt.microsecond)
```

Save and run it:

OBSERVE: Results from Running time_attributes.py

```
2012
10
31
12
30
59
300
```

Finishing the birthday counter

We now have enough information to finish the **birthday.py** program and test it adequately. Let's expand the unittest to properly test the **birthday_counter()** function.

CODE TO EDIT: test_birthday.py

```
from datetime import datetime
import unittest


from birthday import *

class TestBirthday(unittest.TestCase):

    def test_birthday_counter(self):
        ~~self.assertEqual(birthday_counter("10-31-1948"), 100)~~
        # will fail on October 31
        self.assertTrue(birthday_counter("10-31-1948") > 0)

        # will fail on February 1
        self.assertTrue(birthday_counter("02-01-1999") > 0)

    def test_string_to_date(self):

        self.assertRaises(InvalidDateFormat, string_to_date, "10-32-1948")
        # create a new datetime object from scratch
        datetime_obj = datetime(2012, 10, 31)
        self.assertEqual(datetime_obj, string_to_date("10-31-2012"))

if __name__ == "__main__":
    unittest.main()
```

Now, we'll finish the **birthday_counter()** itself. Because datetime handling can get tricky, we'll include lots of comments and **logging.debug** statements. Once we confirm that the provided birthday is valid, we can construct an upcoming birthday using attributes from your own birthday and the current year. Give it a try:

CODE TO EDIT: birthday.py

```
import logging
from datetime import datetime, timedelta
from optparse import OptionParser

logging.basicConfig(filename='birthday.log',level=logging.DEBUG)

class InvalidDateFormat(Exception):
    pass

def string_to_date(date):
    """
    Converts 'MM-DD-YYYY' to a date/time object
        or throws an InvalidDateFormat exception
    """
    try:
        # create a datetime object from the date value
        formatter_string = "%m-%d-%Y"
        birthday = datetime.strptime(date, formatter_string)
    except ValueError as e:
        # log the format error then raise it again so it can be handled gracefully
        logging.error(e)
        raise InvalidDateFormat(e)
    return birthday

def birthday_counter(birthday):
    """
    Returns the number of days until your birthday.
    ~~(not yet fully implemented)~~
    """
    ~~return 100~~
    now = datetime.now()
    birthday = string_to_date(birthday)
    logging.debug("birthday: %s" % birthday)

    # construct the upcoming birthday from this year, your birthday month, and birthday day
    upcoming = datetime(now.year, birthday.month, birthday.day)
    logging.debug("upcoming: %s" % upcoming)

    # Make sure that upcoming is in the future, not the past
    if upcoming < now:
        upcoming = upcoming + timedelta(365)
        logging.debug("fixed upcoming: %s" % upcoming)

    # create a timedelta (duration) between the now and your birthday
    duration = upcoming - now
    logging.debug("duration: %s" % duration)

    # return only the days
    return duration.days

if __name__ == '__main__':
    parser = OptionParser()
    parser.add_option('-b', '--birthday', dest="birthday", action="store",
    help="Your birthday in MM-DD-YYYY format")
    (options, args) = parser.parse_args()

    format_error_message = "birthday.py requires a date in MM-DD-YYYY format"
    if not options.birthday:
        parser.error(format_error_message)

    try:
        print(birthday_counter(options.birthday))
    except InvalidDateFormat:
        parser.error(format_error_message)
```

You'll need to set the Run Configuration for **birthday.py**, as learned earlier, to prompt for the argument **${string_prompt}**. For detailed instructions, see the beginning of the previous lesson.

▶ Save both programs and run the test:

OBSERVE: Results from Running test_birthday.py

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.172s

OK
```

Once the tests pass, run the program:

birthday.py -b 11-01-1957 (as done on 11-29-2010)

```
336
```

So how many days is it until *your* birthday?

Summary

Handling basic dates and times seems easy for us humans to do in our head because we've been taught from a very young age how to read clocks. However, as soon as you need to calculate adding 156 minutes to the current time or 65 days to the current day, things get very challenging. We often need to stop and think about things because the math is not clear—we are converting from decimal into a chaotic mix of base 60, base 24 and other counting systems. Because of this lack of clarity, we need to take extra special care when writing any kind of date/time code.

When you finish the lesson, return to the syllabus and complete the quiz(zes) and project(s).