Python 2: Getting More out of Python

Introduction to Eclipse

Welcome to the O'Reilly School of Technology's (OST) **Getting More Out of Python** course! We're happy you've chosen to learn Python programming with us. By the time you finish the course, you will have expanded your knowledge of Python and applied it to some really interesting technologies.

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 lik
e 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**, 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.

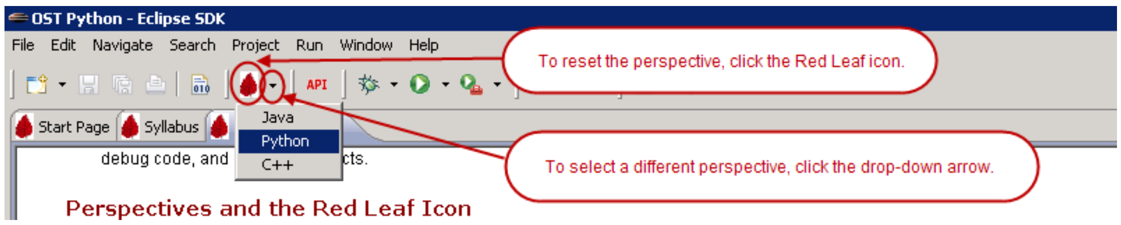
Perspectives and the Red Leaf Icon

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.

The Red Leaf icon has these functions:

- **To reset the current perspective:** click the icon.
- **To change perspectives:** click the drop-down arrow beside the icon to select different perspectives designed for each course that uses Ellipse.
- **To 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, you would select **Python** (it should already be selected for you now):


OST Python - Eclipse SDK
File Edit Navigate Search Project Run Window Help
API
Start Page
Syllabus
Java
Python
C++
debug code, and
cts.
Perspectives and the Red Leaf Icon

To reset the perspective, click the Red Leaf icon.

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

Note If you select a Red Leaf icon option, it will reset your session, so you'll need to reselect this course and reopen this lesson.

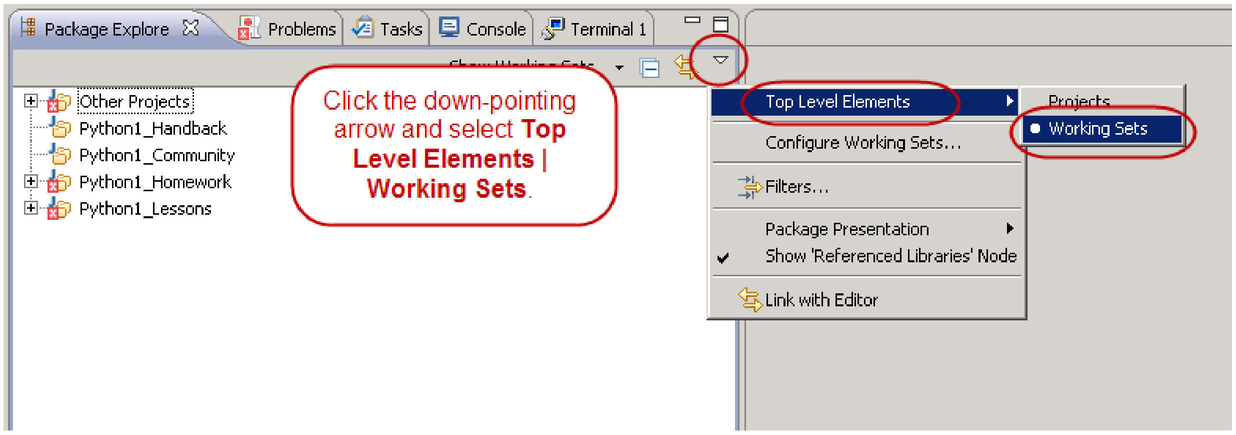
Working Sets

You'll use *working sets* for the 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.

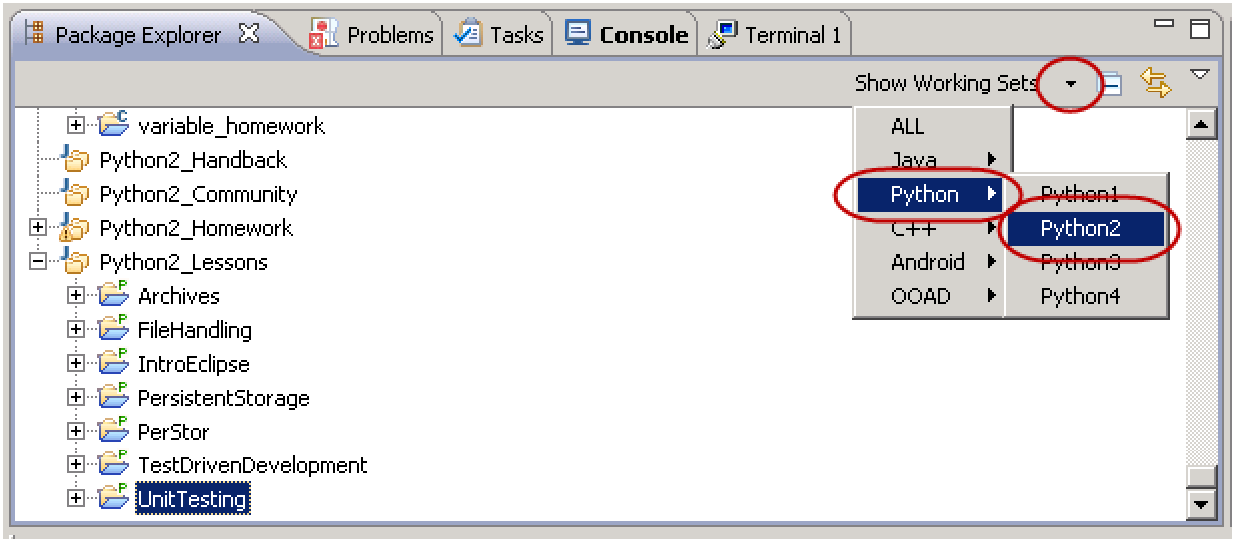
A working set is a convenient way to group related items together. You can assign a project to one or more working sets. In some cases, like the Python extension to Eclipse, new projects are created in a catch-all "Other Projects" working set. To better organize your work, we'll have you assign your 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 now, make sure your working sets are displayed by clicking the down-pointing arrow on the top right of the Package Explorer window, and select **Top Level Elements | Working Sets**:



Then, click the down-pointing arrow next to **Show Working Sets** and select **Python | Python 2**:



Programming in Python with Eclipse

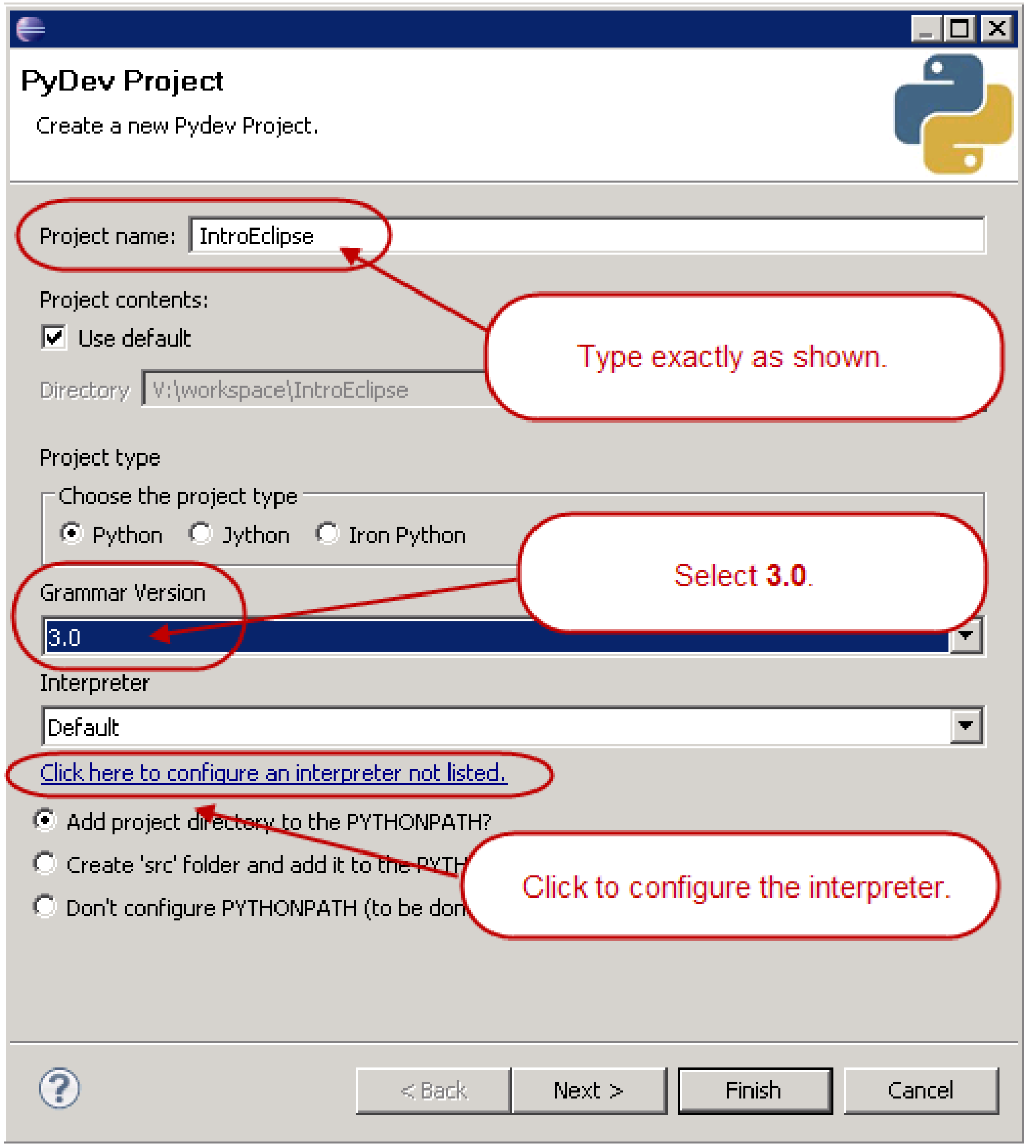
A First Program

When learning a new language in computer programming, it is traditional to use the words "hello world" as your first example. Unfortunately, since "hello world" can be written in a single line, that doesn't make for a great example in Python. Instead, we'll look at a slightly more complicated example that not only prints "hello" and "goodbye," but also does a little calculation on the way.

Let's set up an environment for our first file. In Eclipse, all files must be within *projects*. A project is a container that holds resources (such as source code, images, and other things) needed to build a piece of software. We're going to make a project named **IntroEclipse**. Please use that exact name, with the same capitalization.

In creating a new project, you'll need to read ahead a few steps because once the dialog box appears, you will not be able to return to the Lesson until finishing it. You can also view the PDF version of the course in another window while creating the project. This is the only time you should need to work in a separate window in this course.

Now, let's create a **PyDev project** in Eclipse. (PyDev is the name of the Eclipse add-in that adapts it to handle Python). To start a new project, select the menu item **File | New | PyDev Project**. Enter the name **IntroEclipse**, select **3.0** for the Grammar Version, and click the link to configure an interpreter:


PyDev Project

Create a new Pydev Project.

Project name: IntroEclipse

Project contents:

[✓] Use default

Directory V:\workspace\IntroEclipse

Project type

Choose the project type

[● Python] ○ Jython ○ Iron Python

Grammar Version

3.0

Interpreter

Default

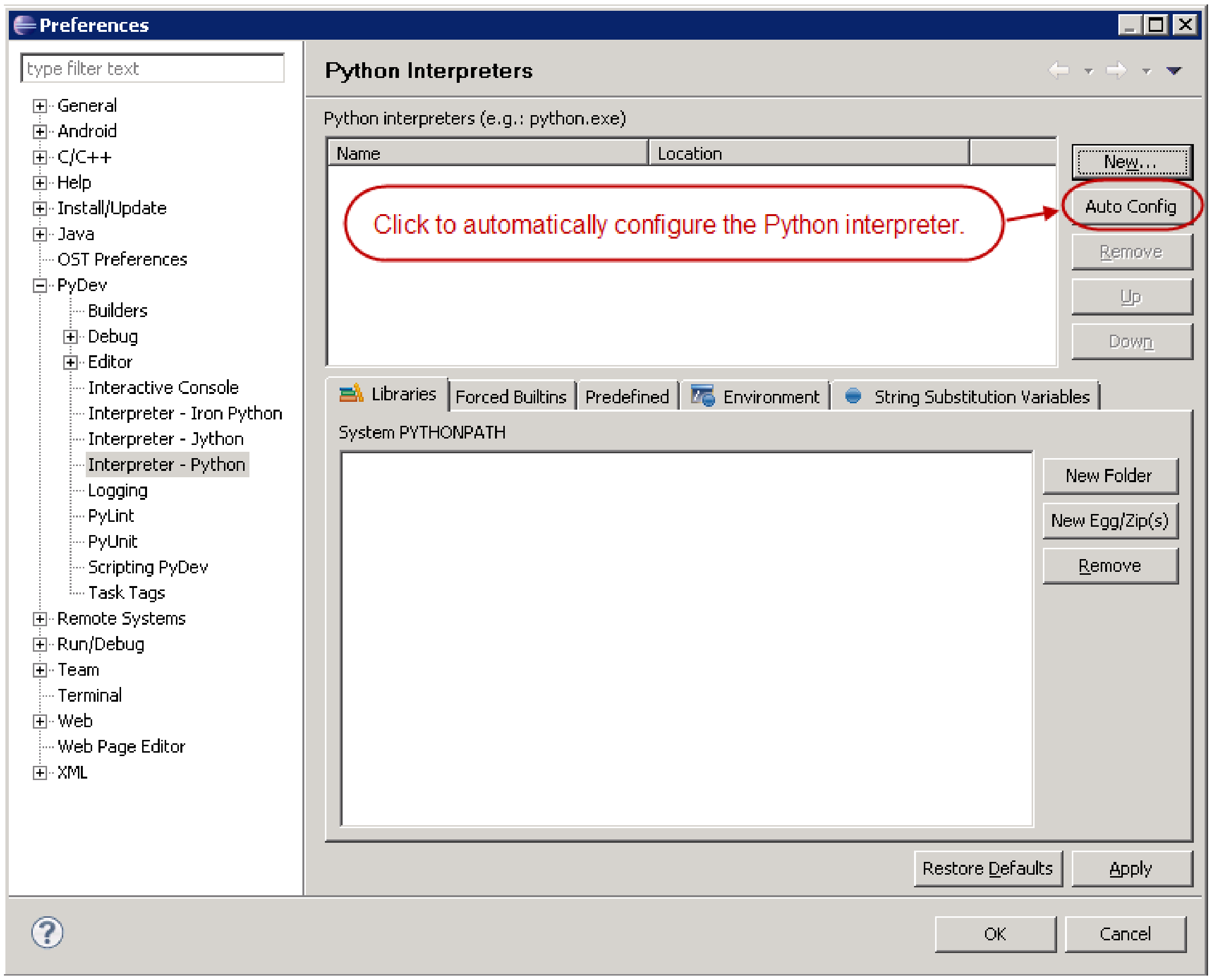
Click here to configure an interpreter not listed.

[● Add project directory to the PYTHONPATH?]

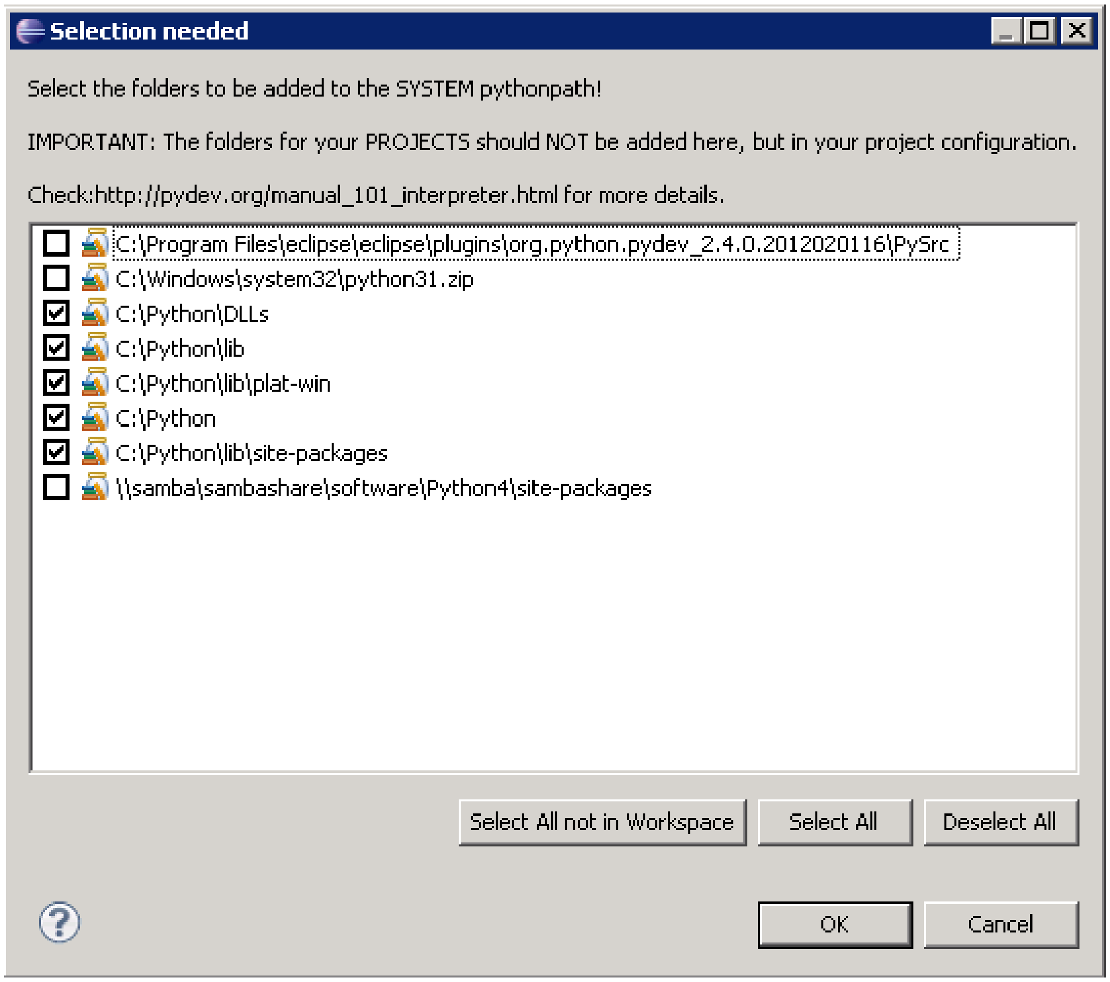
○ Create 'src' folder and add it to the PYTH

○ Don't configure PYTHONPATH (to be don

On the Preferences screen, click **Auto Config** to configure the Python interpreter:



A Selection Needed screen appears. Click **OK** to select the default settings:



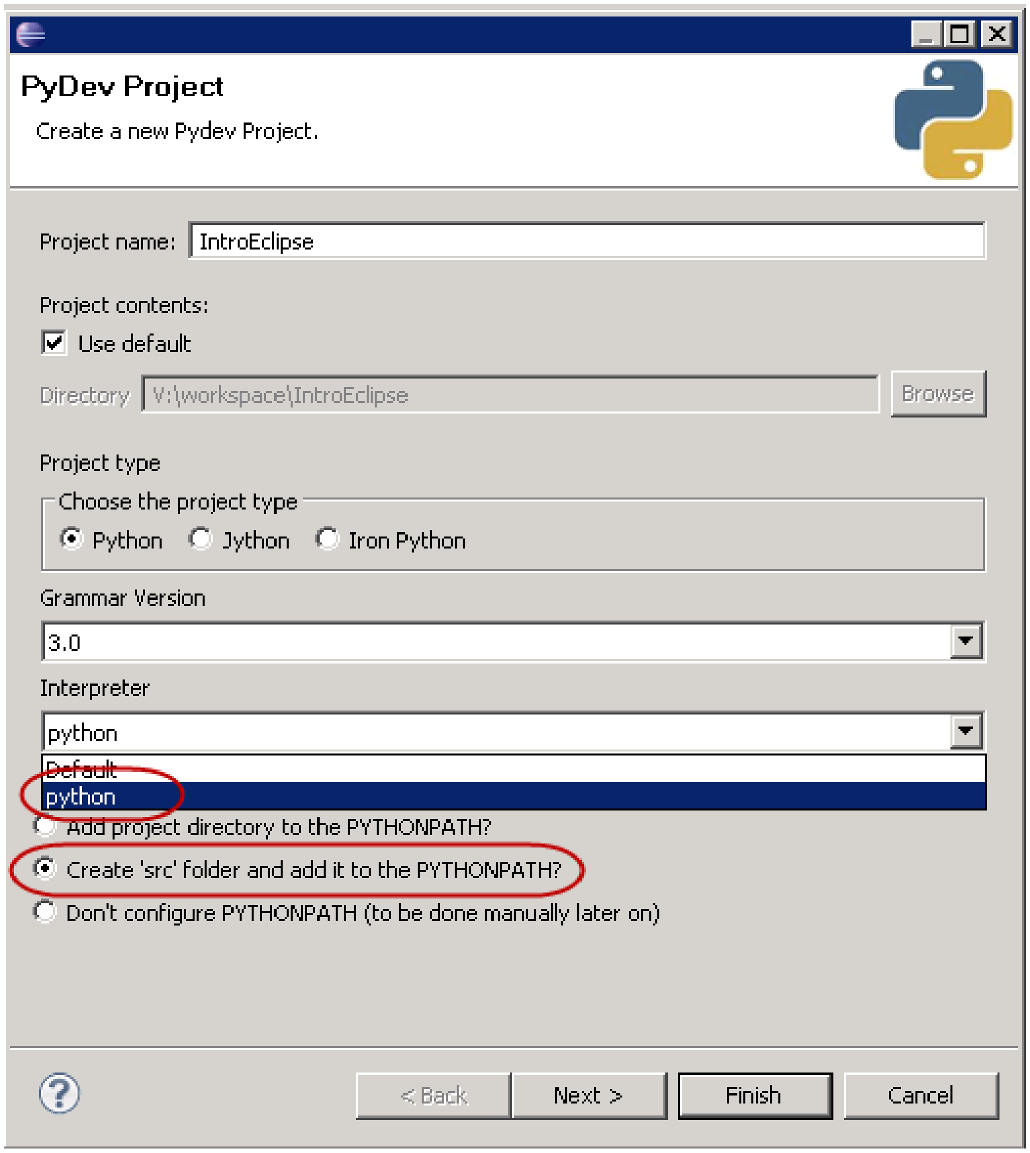
Select the folders to be added to the SYSTEM pythonpath!

IMPORTANT: The folders for your PROJECTS should NOT be added here, but in your project configuration.

Check:http://pydev.org/manual_101_interpreter.html for more details.

- [ ] C:\Program Files\eclipse\eclipse\plugins\org.python.pydev_2.4.0.2012020116\PySrc
- [ ] C:\Windows\system32\python31.zip
- [x] C:\Python\DLLs
- [x] C:\Python\lib
- [x] C:\Python\lib\plat-win
- [x] C:\Python
- [x] C:\Python\lib\site-packages
- [ ] \\samba\sambashare\software\Python4\site-packages

Click **OK** again to return to the Pydev Project screen. Select the **python** interpreter we just created, and make sure **Create 'src' folder and add it to the PYTHONPATH' is selected**:


PyDev Project

Create a new Pydev Project.

Project name: IntroEclipse

Project contents:

✓ Use default

Directory V:\workspace\IntroEclipse

Project type

Choose the project type

● Python ○ Jython ○ Iron Python

Grammar Version

3.0

Interpreter

python

Default

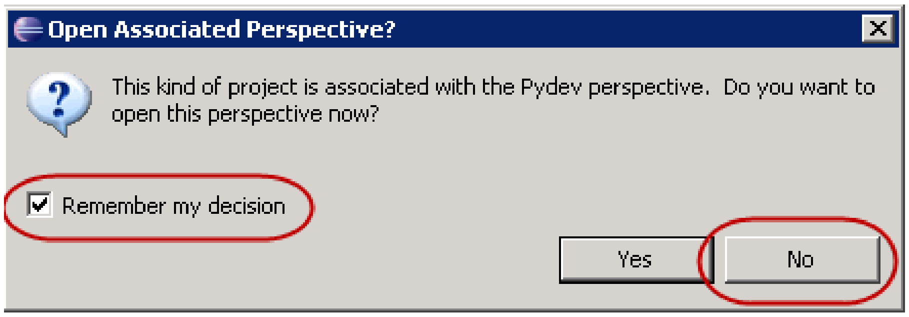
python

○ 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)

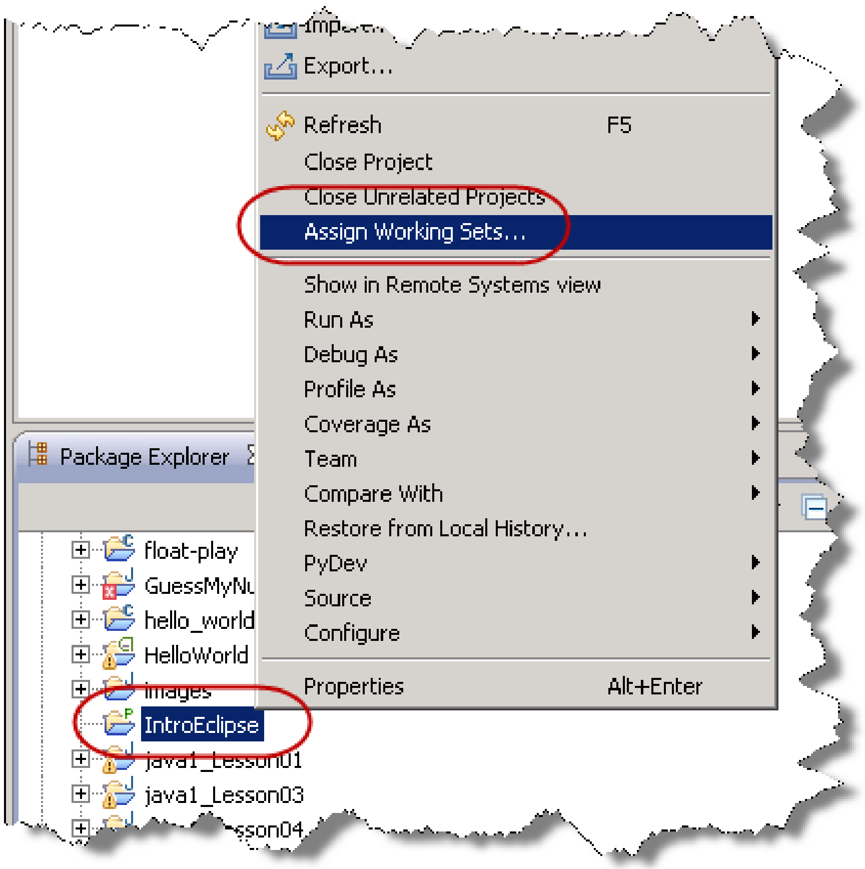
Click **Finish**. You see a prompt to change perspectives. Check the **Remember my decision** box and click **No**:


Open Associated Perspective?

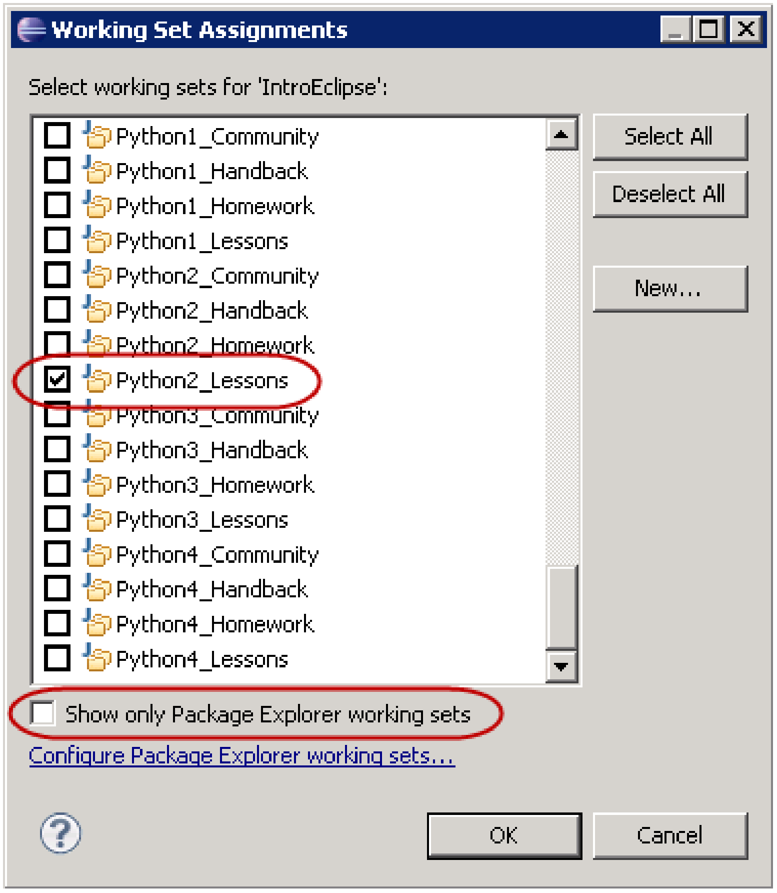
This kind of project is associated with the Pydev perspective. Do you want to open this perspective now?

✓ Remember my decision

When you first create a PyDev project, it is placed in the **Other Projects** working set. You'll want to keep your Python projects together, so go ahead and put your newly created project into the **Python2_Lessons** working set. Select the **IntroEclipse** project. Right-click it and select **Assign Working Sets...**:



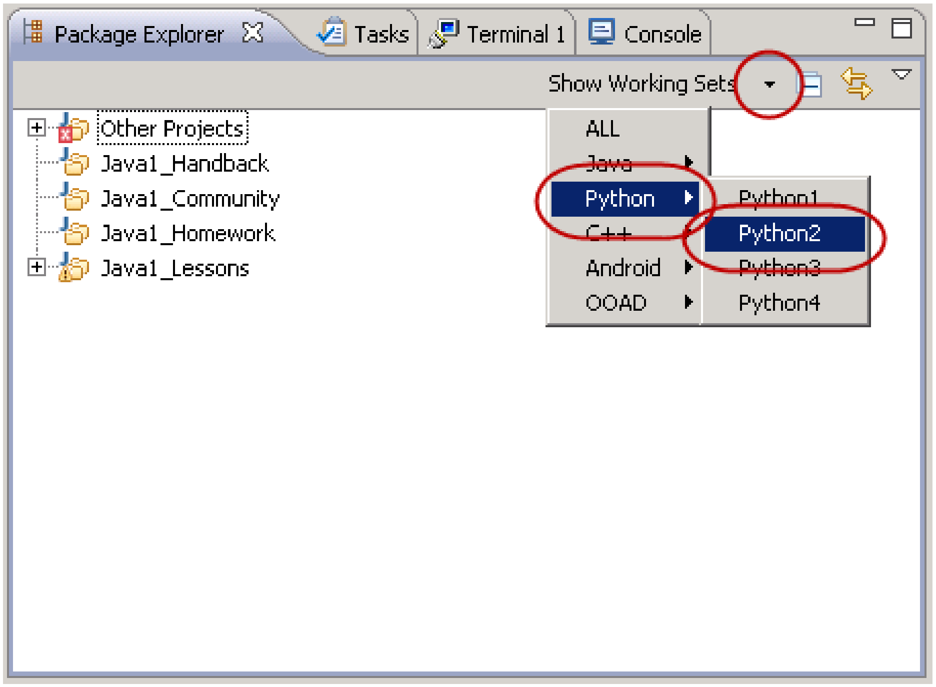
The Working Set Assignments screen appears. Click **Deselect All** to clear any selected working sets, and then check the box for the **Python2_Lessons** working set (the one for this course), UNcheck the **Show only Package Explorer working sets** box, and click **OK**:



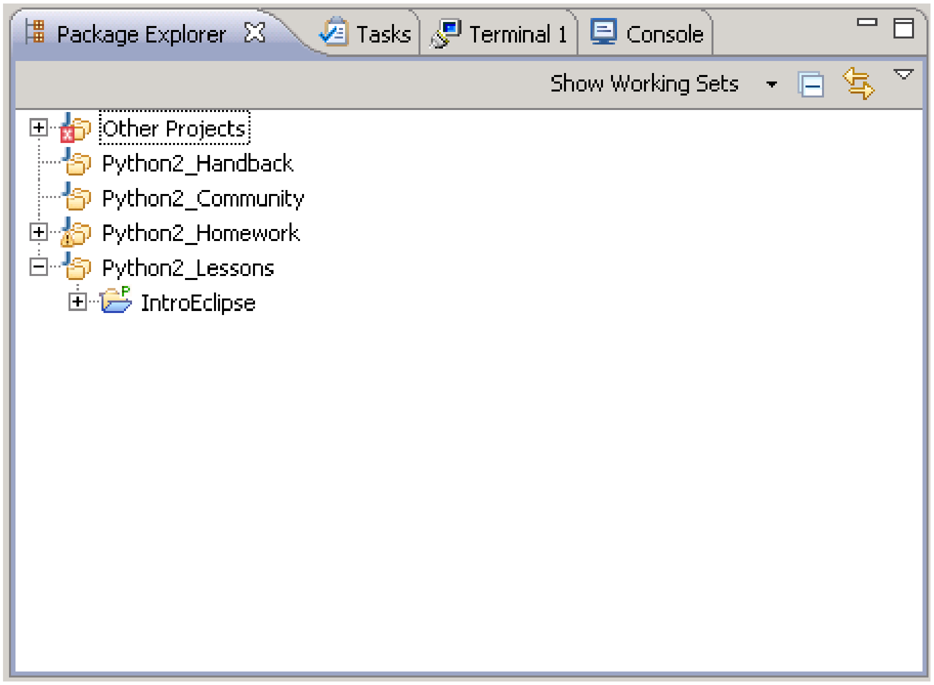
Click **OK** when you finish. You will need to do this for each new project you create.

Note You might not see as many working sets; you'll only see ones for courses you're enrolled in.

To see the projects in your **Python2_Lessons** working set in the Package Explorer panel, click the downward-pointing arrow next to the **Show Working Sets** button, and select **Python | Python2**:



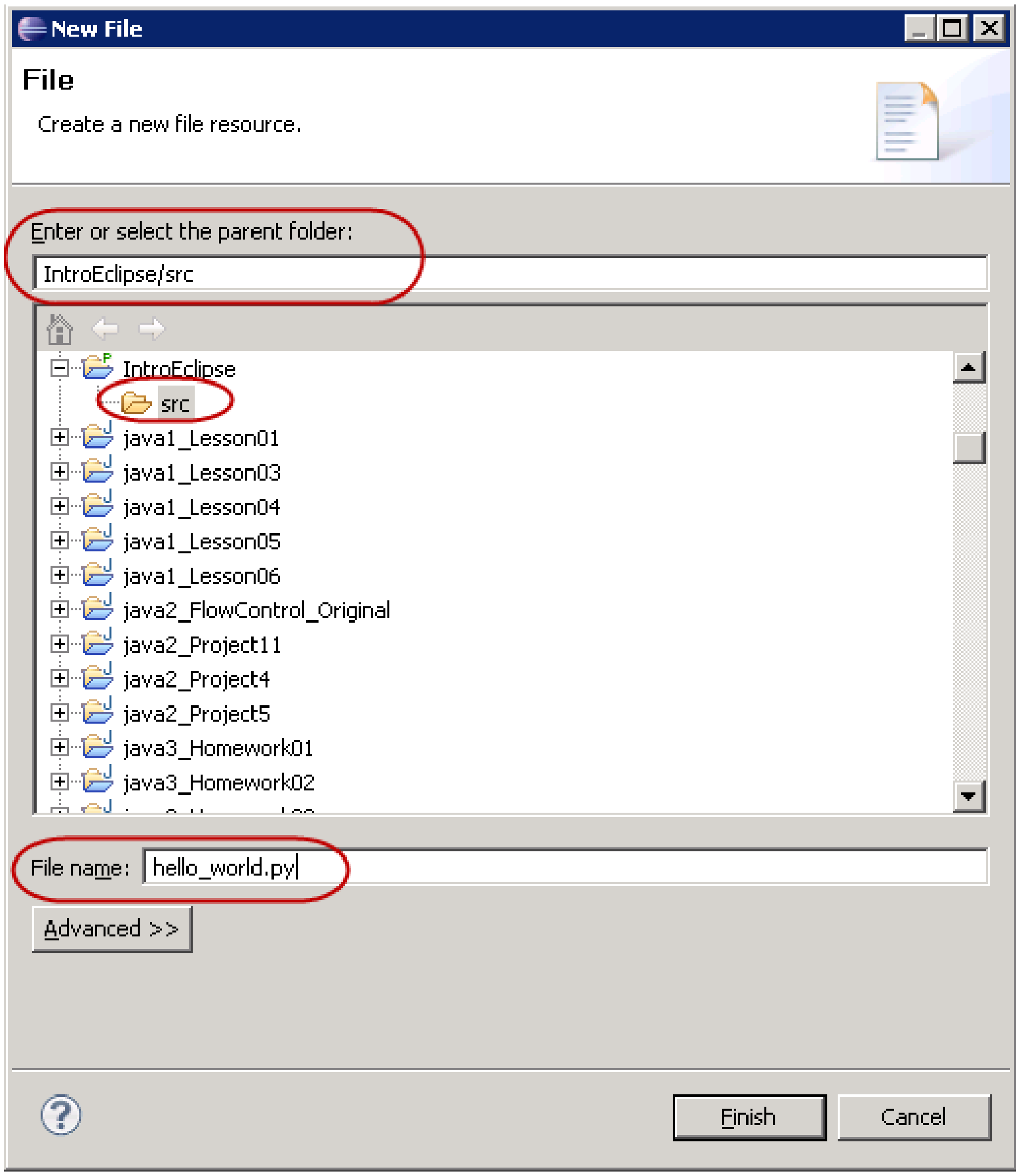
Now you should see your **IntroEclipse** project listed in the **Python2_Lessons** working set in the **Package Explorer** panel on the lower left corner of your Eclipse screen:



This hierarchical view of the resources (directories and files) in Eclipse is commonly called the *workspace*. You now have a *project* called IntroEclipse in your workspace.

Before you go on, make sure that the IntroEclipse project is displayed in the Package Explorer window.

Right-click your **IntroEclipse** project in the Package Explorer, and select **New | File**. A New File dialog box appears. Select the **src** subdirectory of **IntroEclipse**, enter the filename **hello_world.py**, and then click **Finish**:



A new editor window appears next to the workspace. You'll edit your code in this window because it understands Python syntax.

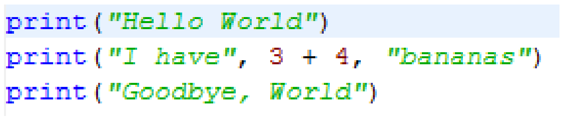
Enter the blue code below into the editor window:

Note When you enter an opening parenthesis, Eclipse automatically adds the closing parenthesis.

CODE TO TYPE:

```
print("Hello World")
print("I have", 3 + 4, "bananas")
print("Goodbye, World")
```

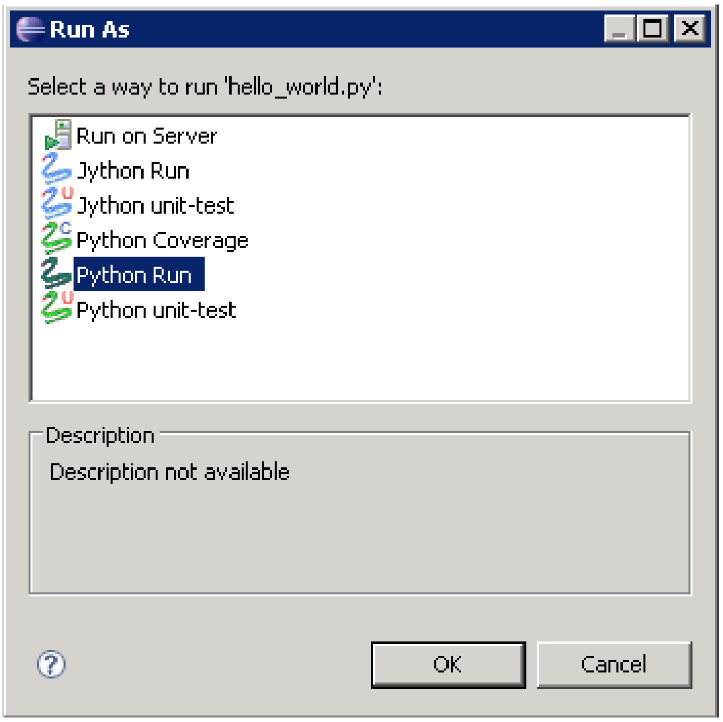
Your code should look like this:



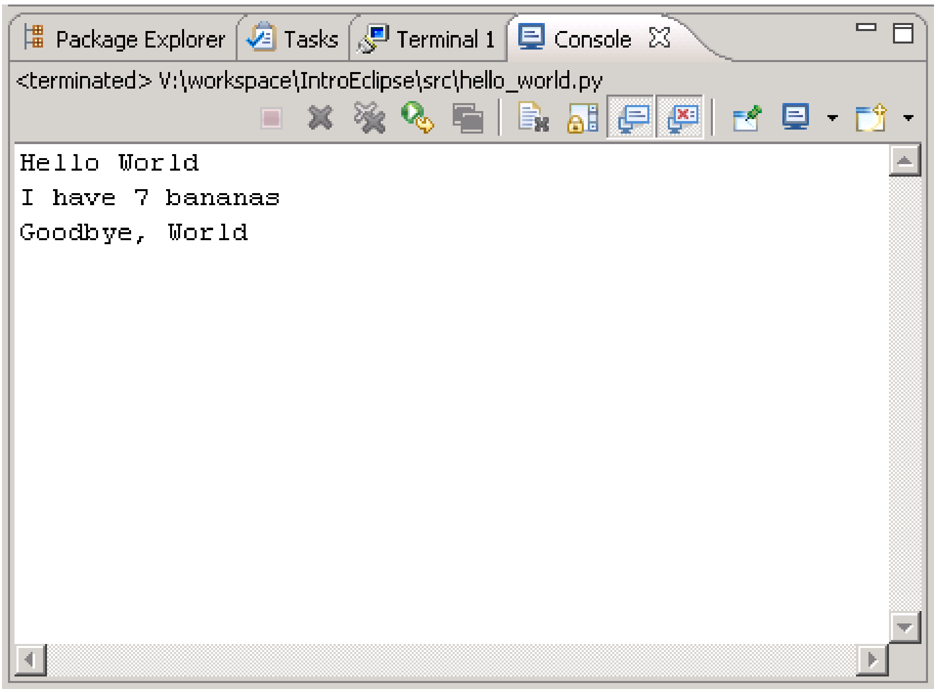
```
print("Hello World")
print("I have", 3 + 4, "bananas")
print("Goodbye, World")
```

Save it. In the top Eclipse menu bar (not the O'Reilly tab bar) choose **File | Save** or click the **Save** icon at the top of the screen: 💾 (we'll show that icon from now on when we want you to save a file).

Now choose **Run | Run** from the top menu bar (if you don't see this menu choice, click in hello_world.py in the Editor Window again). You can also click the run icon: . From now on, when we want you to save AND run a program, we'll show that icon. The first time you run a program, you'll see this prompt:



Select **Python Run**. If you entered the code correctly, you'll see that the workspace switches to the Console view and displays the output from your very first Python program:



```
<terminated> V:\workspace\IntroEclipse\src\hello_world.py
```

```
Hello World
I have 7 bananas
Goodbye, World
```

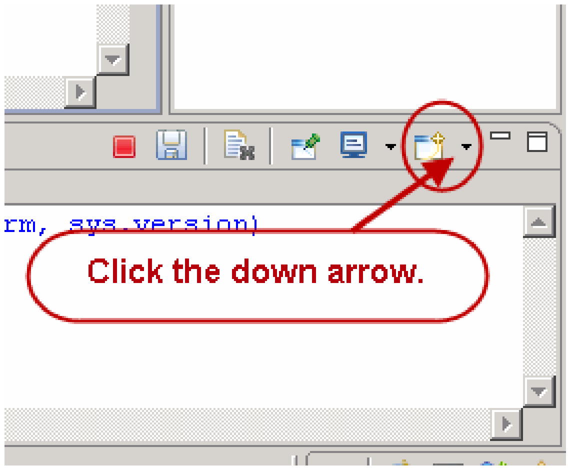
Congratulations! You're officially a Python programmer! Of course this program isn't very complex, but the interpreter has done the calculation you asked it to do. Pat yourself on the back! You're off to a strong start. Experiment with other calculations. You can probably work out how to save modified programs under different names (Hint: **File | Save As**).

The Interactive Interpreter

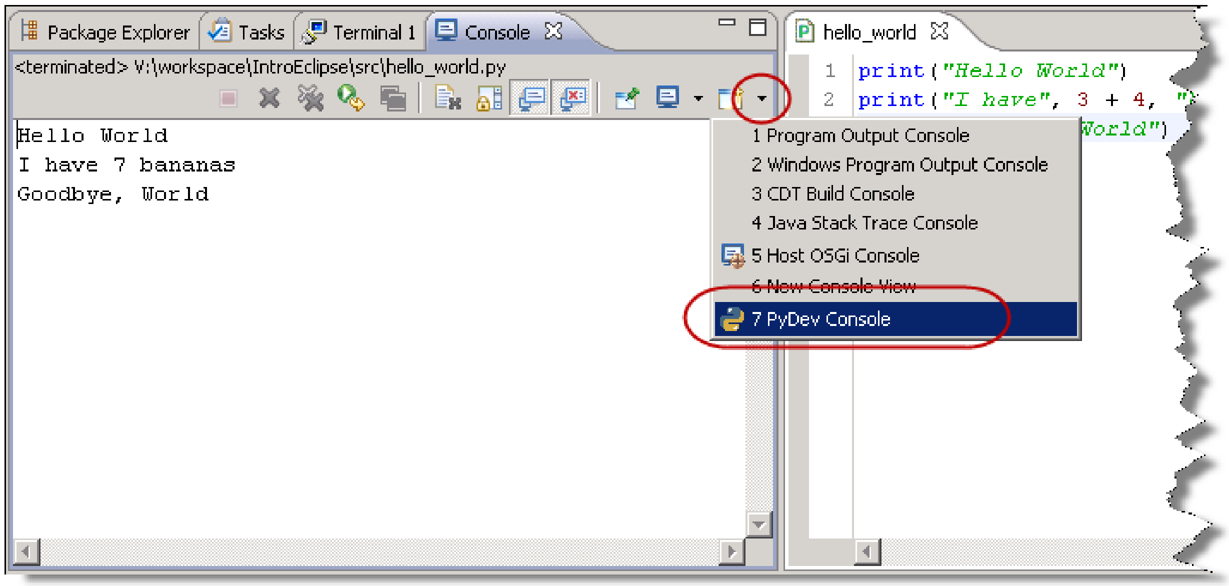
In Python you can run the interpreter in interactive mode when you want to try things out, and see results printed right away. That instant feedback is really handy when you're learning a new language.

Eclipse gives you access to interactive Python consoles.

Select the **Console** tab in the workspace window, and click the down arrow to the right of the **Open Console** icon:



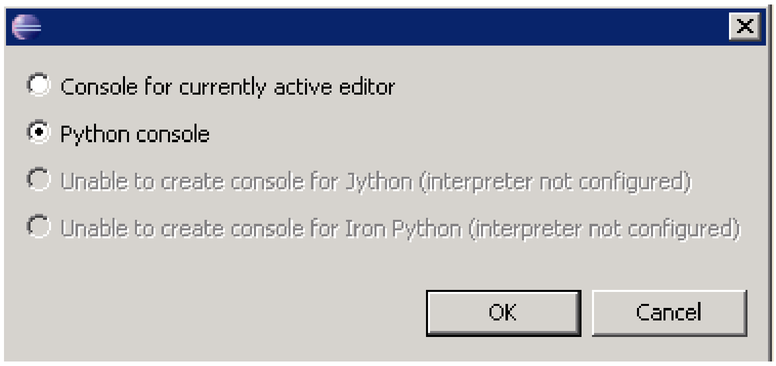
Select **Pydev Console** from the pull-down menu:



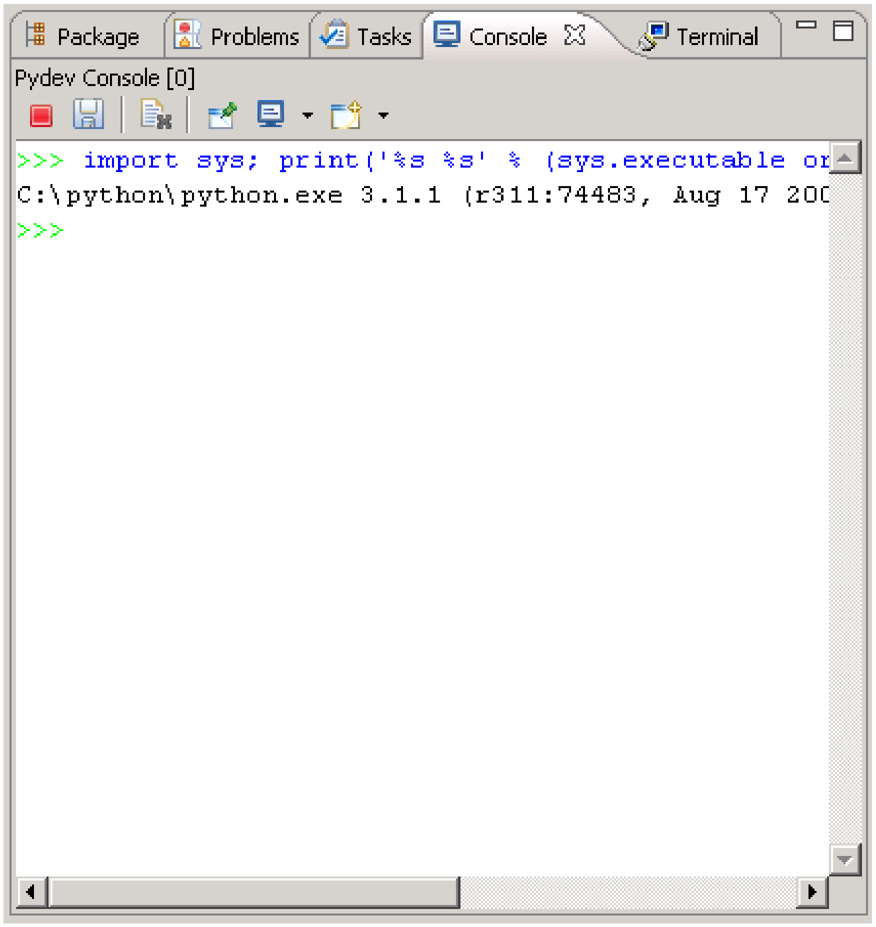
```
<terminated> V:\workspace\IntroEclipse\src\hello_world.py
Hello World
I have 7 bananas
Goodbye, World
```

```
print("Hello World")
print("I have", 3 + 4, "
```

Select a Python console:


Console for currently active editor
Python console
Unable to create console for Jython (interpreter not configured)
Unable to create console for Iron Python (interpreter not configured)
OK
Cancel

A new console appears, with the interactive prompt **>>>**. The console is ready for your input:



```
>>> import sys; print('%s %s' % (sys.executable or
C:\python\python.exe 3.1.1 (r311:74483, Aug 17 200
>>>
```

If you enter one of the lines from the program you just ran, the output will appear in the console window. This interactive interpreter window allows you to enter both statements and expressions (we'll cover those in detail later). Statements are executed pretty much as if they were part of a program; the expressions are evaluated and the resulting value is printed (as long as you're in interactive mode).

Type the code in blue below in the PyDev Console window. **(When we say TYPE the code, do it. It's good for you!)** The interpreter prints a result for each expression. (You'll see a different prompt after the fourth line. We'll talk about that in a minute):

CODE TO TYPE:

```
>>> "hello" + " world"
'hello world'
>>> 'hello' + ' world'
'hello world'
>>> """hello""" + ''' world'''
'hello world'
>>> """hello
... world"""
'hello\nworld'
```

So, what happened here? The first three lines are all examples of *string concatenation*— a second string is appended to the first, giving a longer string as a result. Strings can have either single (') or double (") quotation marks around them, and either one quotation mark or three at the beginning and end of the string. Use exactly the same combination at both ends.

The last expression, running over lines 4 and 5 of the input, shows an important difference between the one-quotation mark and the three-quotation mark forms. A string given in one-quotation mark form *must* begin and end on the same line. Three-quotation mark strings can spread across more than one line.

The fourth example actually does extend across two lines, so the interpreter changed its prompt from **>>>** to **...** (ellipses) after you entered the first line. Those ellipses let you know that you've got an incomplete statement or expression in your code. When you completed the string with the second line of input, the interpreter then printed the value of the two-line expression. You can see that the line feed between **hello** and **world** is represented by **\n**, which is known in Python as a string *escape sequence*.

First Hurdle Cleared

Phew! That was a whole lot of introduction there. Thanks for sticking with me. Keep it up, you're doing great so far. See you at the next lesson!

Note	As we mentioned at the beginning, you made some changes to your working environment during this lesson; now, you should exit Eclipse to save those changes and restart it to continue with the homework and additional lessons.

Unit Testing

Welcome to the second course in the O'Reilly School of Technology's Python series!

Course Objectives

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

- demonstrate understanding of Agile processes and test-driven development.
- manage files, persistent storage, archives, and serialization.
- create a Graphical User Interface in Python.
- design and implement relational databases using Python and SQL.
- create and send emails from Python programs.
- build a full-fledged Python database application.

In this course, you'll learn more in-depth techniques and strategies for programming with Python. Using the Ellipse integrated learning environment, you'll get hands-on experience with Python's modular unit testing features; file handling, storage, and archival; graphical user interfaces; and technologies for working with databases and email.

unittest

Your first lesson in Python 2 picks up where we left off in the Python 1 course, focused on debugging programs. Here you'll learn about the second, and more widely used, built-in Python testing framework, *unittest*. Unittest is a more formal testing framework, which can be integrated with existing uses of *doctest*, if necessary.

Assertions

An important statement contained within Python that you haven't come across before is the *assert* statement. The syntax for this statement is:

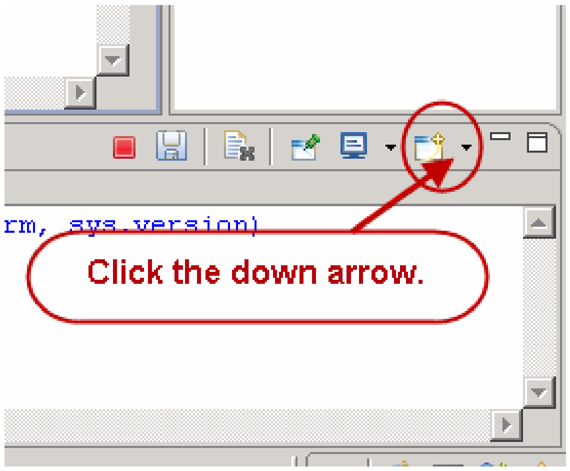
OBSERVE: assert statement syntax

```
assert condition[, message]
```

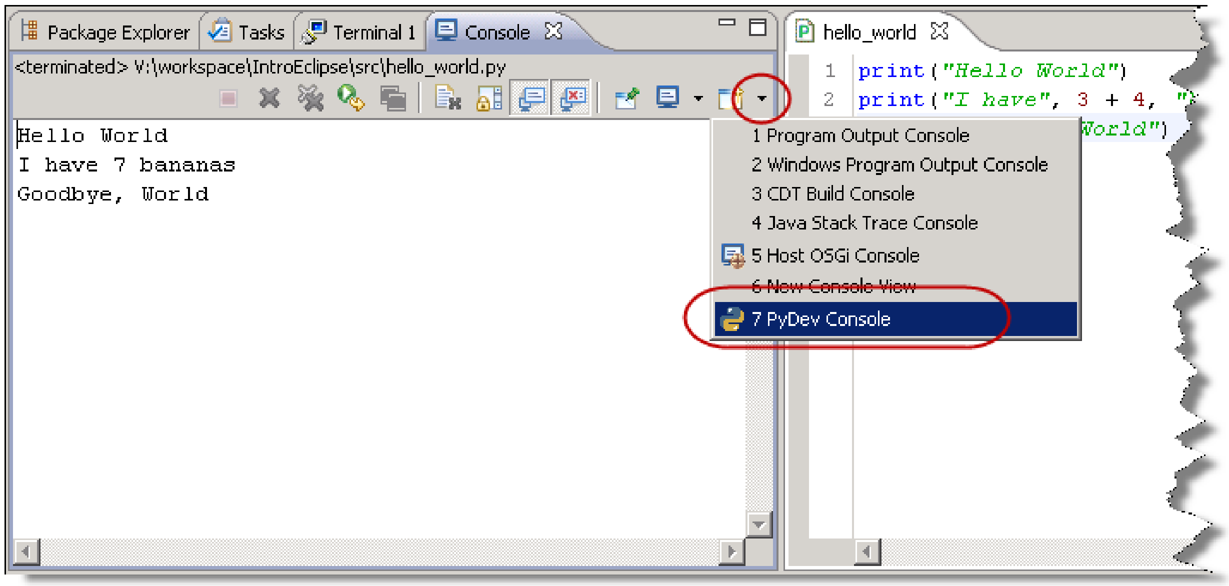
In the assert statement, the **condition** is tested, and if it evaluates false, an AssertionError exception is raised. If there's a **message**, it is printed with the AssertionError. Let's try using the assert statement right now in an interactive console window.

In case you've forgotten how to get to the interactive console, here's how to do it:

Select the **Console** tab in the workspace window, and click the down arrow to the right of the **Open Console** icon:



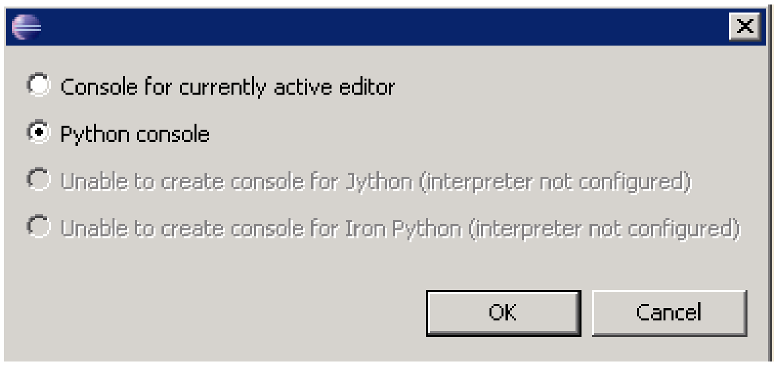
Select **Pydev Console** from the pull-down menu:



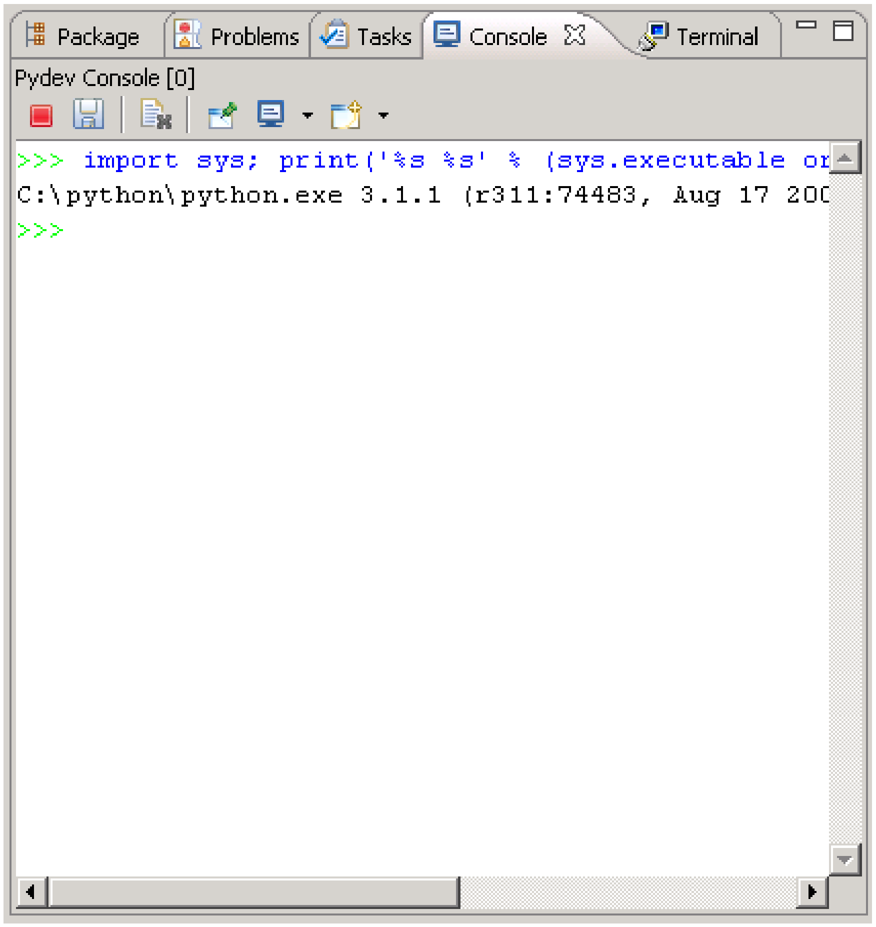
```
<terminated> V:\workspace\IntroEclipse\src\hello_world.py
Hello World
I have 7 bananas
Goodbye, World
```

```
print("Hello World")
print("I have", 3 + 4, "
```

Select **Python console**:


Console for currently active editor [✓ Python console] Unable to create console for Jython (interpreter not configured) Unable to create console for Iron Python (interpreter not configured)

A new console appears, with the interactive prompt **>>>**. The console is ready for your input:



```
>>> import sys; print('%s %s' % (sys.executable or
C:\python\python.exe 3.1.1 (r311:74483, Aug 17 200
>>>
```

Type this code into the interactive interpreter console:

INTERACTIVE SESSION:

```
>>> assert 1 == 1
>>> assert 1 == 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 1 == 2, "One isn't two and the universe is still rational"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: One isn't two and the universe is still rational
>>>
```

We use assert statements in our programs, to assert conditions that we believe must always be true. If we are correct, the program runs as expected. But, if a programming error or mistaken assumption invalidates the condition, Python will let us know, usually early in the life of the program.

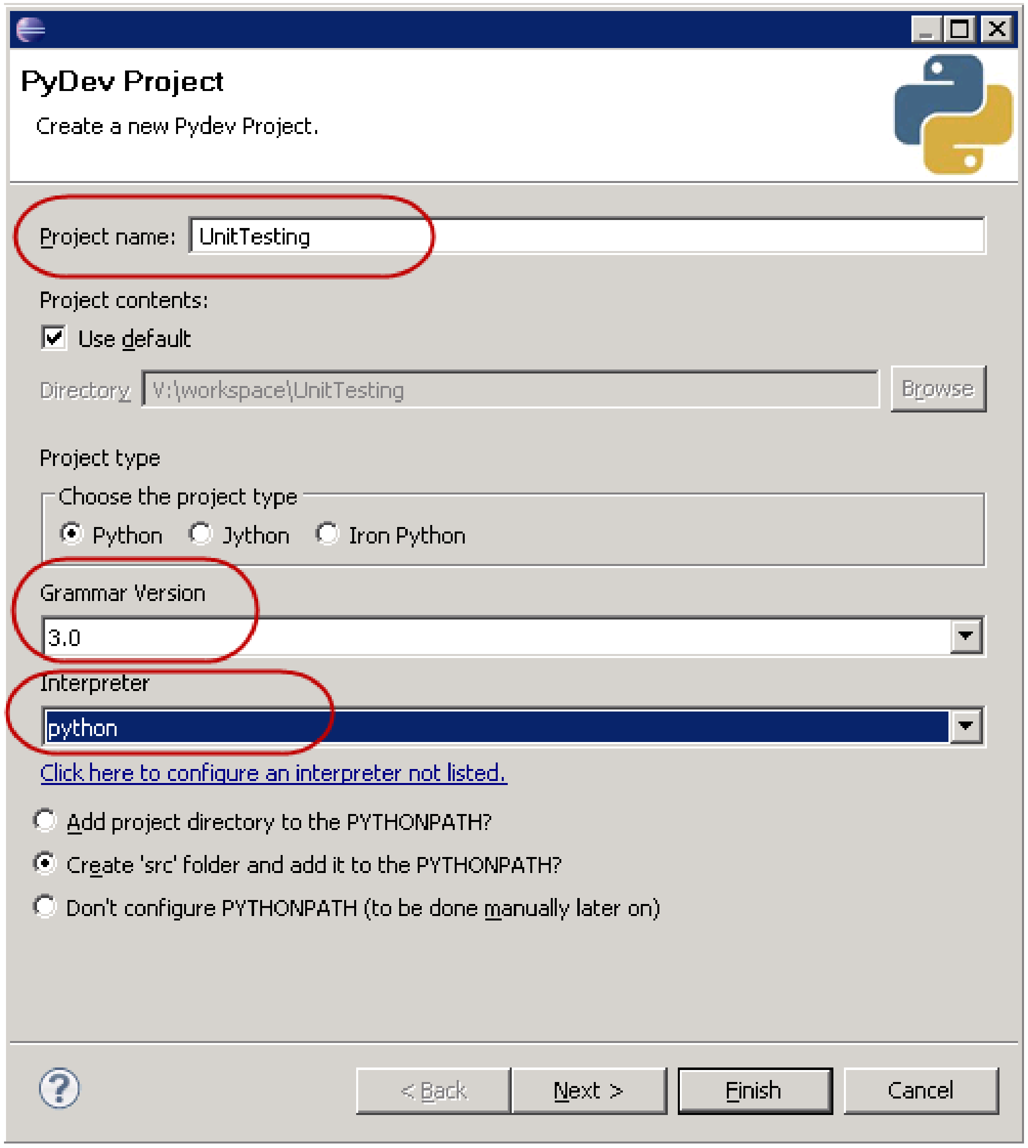
AssertionError exceptions are handled using the unittest module. To write tests, we create *test cases* that are subclasses of the **unittest.TestCase** class. Our subclasses can use the methods defined by the superclass. Many of those methods' names begin with the prefix "assert." By calling these methods, you have the test case make assertions about your program in a controlled environment. Any AssertionErrors that arise are handled by the framework and reported as a failure of the associated test. Other exceptions are regarded as errors.

A Basic unittest Example

For our first example, we'll use the **square()** method from our **testable.py** code that we created in the "Introduction to Python" course. Our goal now is to write code that will cube the values passed. This will allow us to compare the two testing modules.

Just to make sure we're all up to speed, let's review setting up a new PyDev project and a Python program:

To start a new project, select the menu item **File | New | PyDev Project**. Enter the name **UnitTesting**, select **3.0** for the Grammar Version, and select the **python** interpreter (if it's not available, click on the link to configure an interpreter):



PyDev Project

Create a new Pydev Project.

Project name: UnitTesting

Project contents:

[✓] Use default

Directory V:\workspace\UnitTesting Browse

Project type

Choose the project type

(●) Python ( ) Jython ( ) Iron Python

Grammar Version

3.0

Interpreter

python

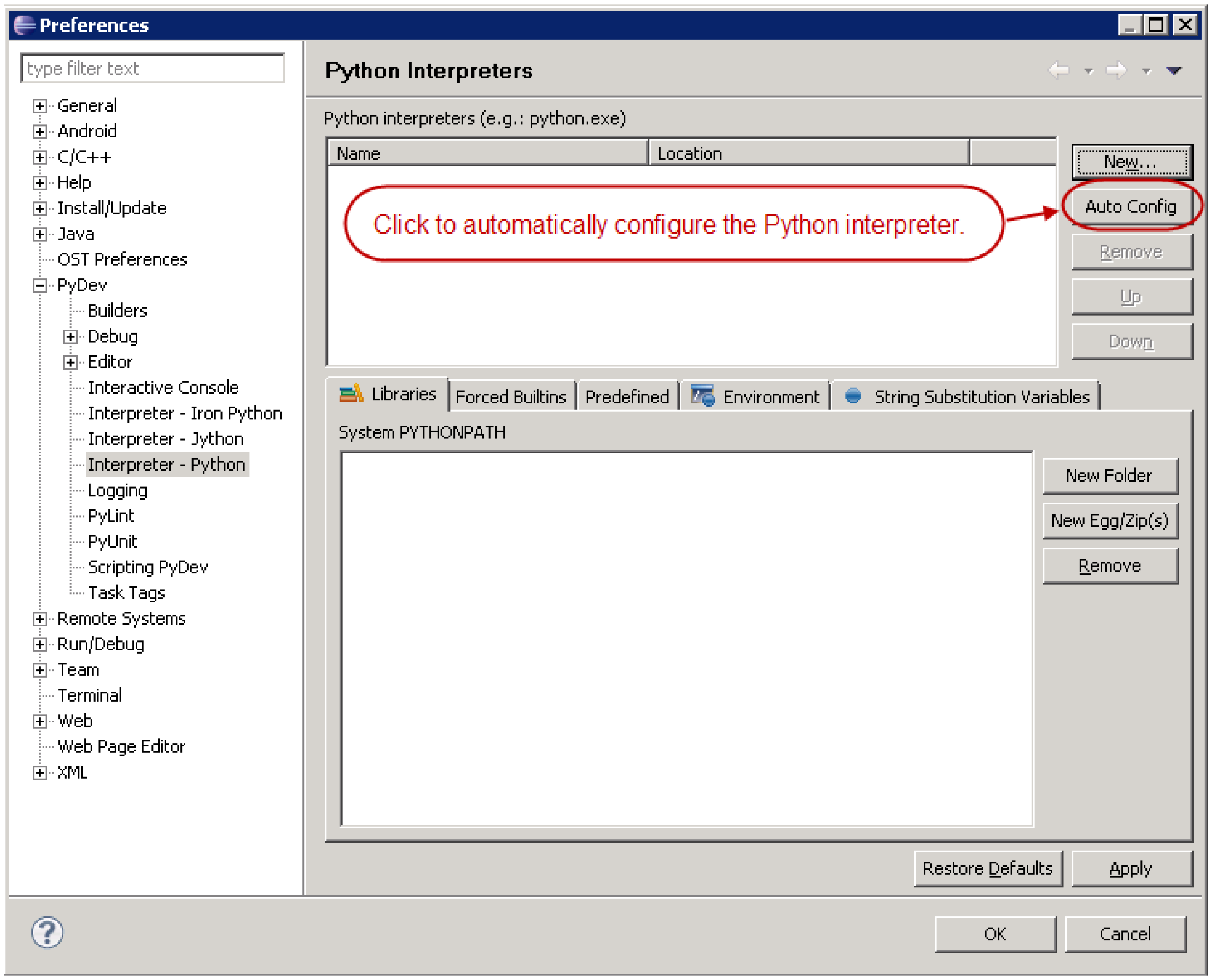
Click here to configure an interpreter not listed.

( ) Add project directory to the PYTHONPATH?

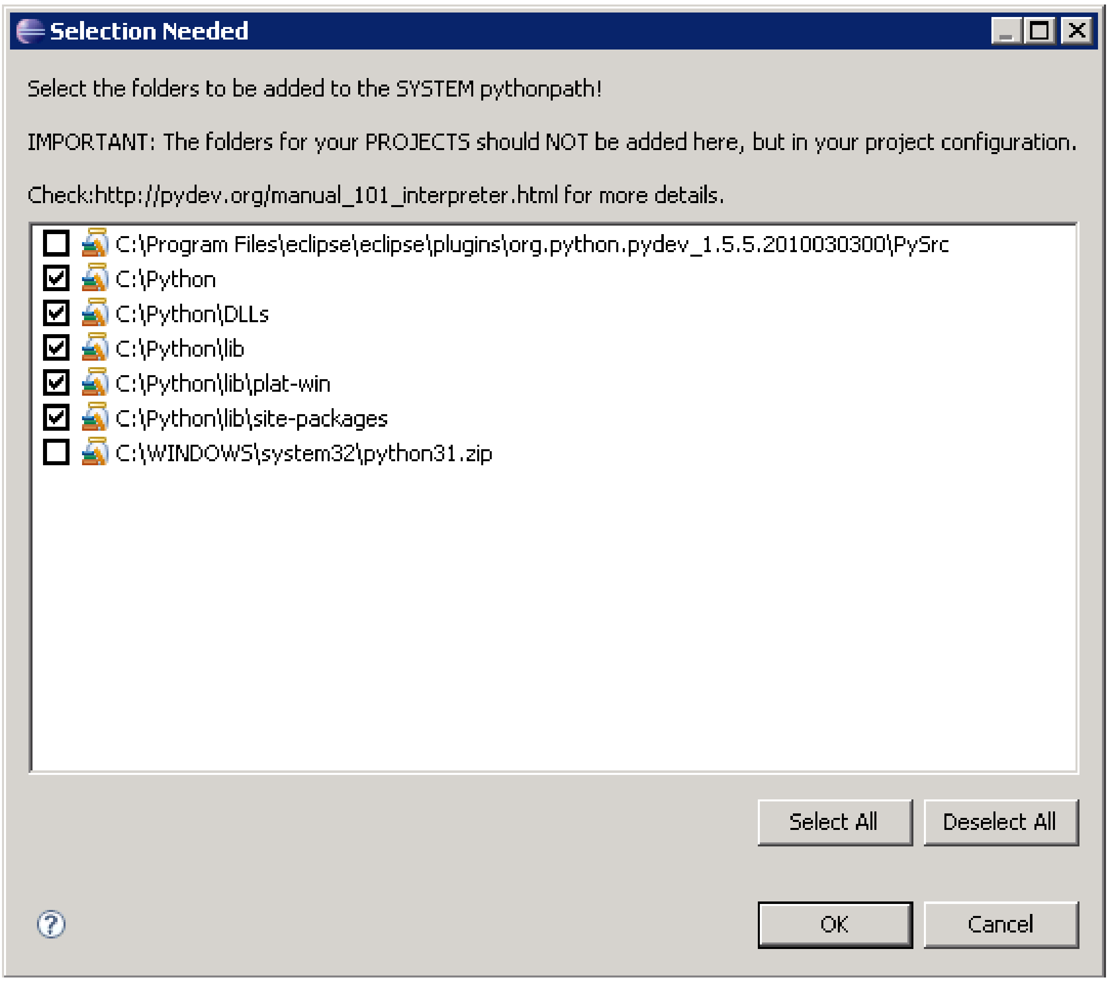
(●) Create 'src' folder and add it to the PYTHONPATH?

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

If you clicked the link to configure the interpreter, go to the Preferences screen now and click **Auto Config**:



A **Selection Needed** screen appears. Click **OK** to select the default settings:


Selection Needed

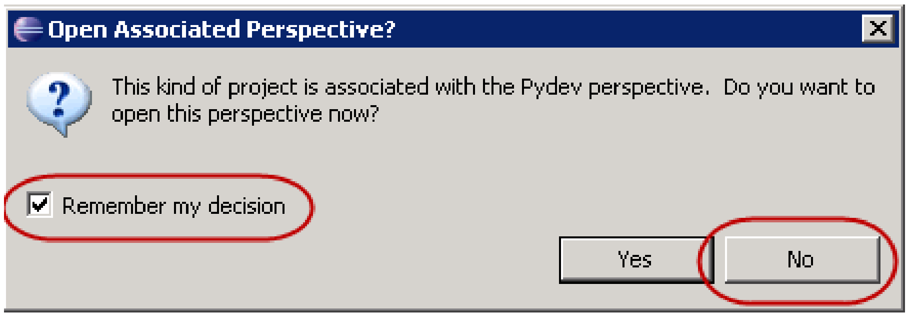
Select the folders to be added to the SYSTEM pythonpath!

IMPORTANT: The folders for your PROJECTS should NOT be added here, but in your project configuration.

Check:http://pydev.org/manual_101_interpreter.html for more details.

- [ ] C:\Program Files\eclipse\eclipse\plugins\org.python.pydev_1.5.5.2010030300\PySrc
- [x] C:\Python
- [x] C:\Python\DLLs
- [x] C:\Python\lib
- [x] C:\Python\lib\plat-win
- [x] C:\Python\lib\site-packages
- [ ] C:\WINDOWS\system32\python31.zip

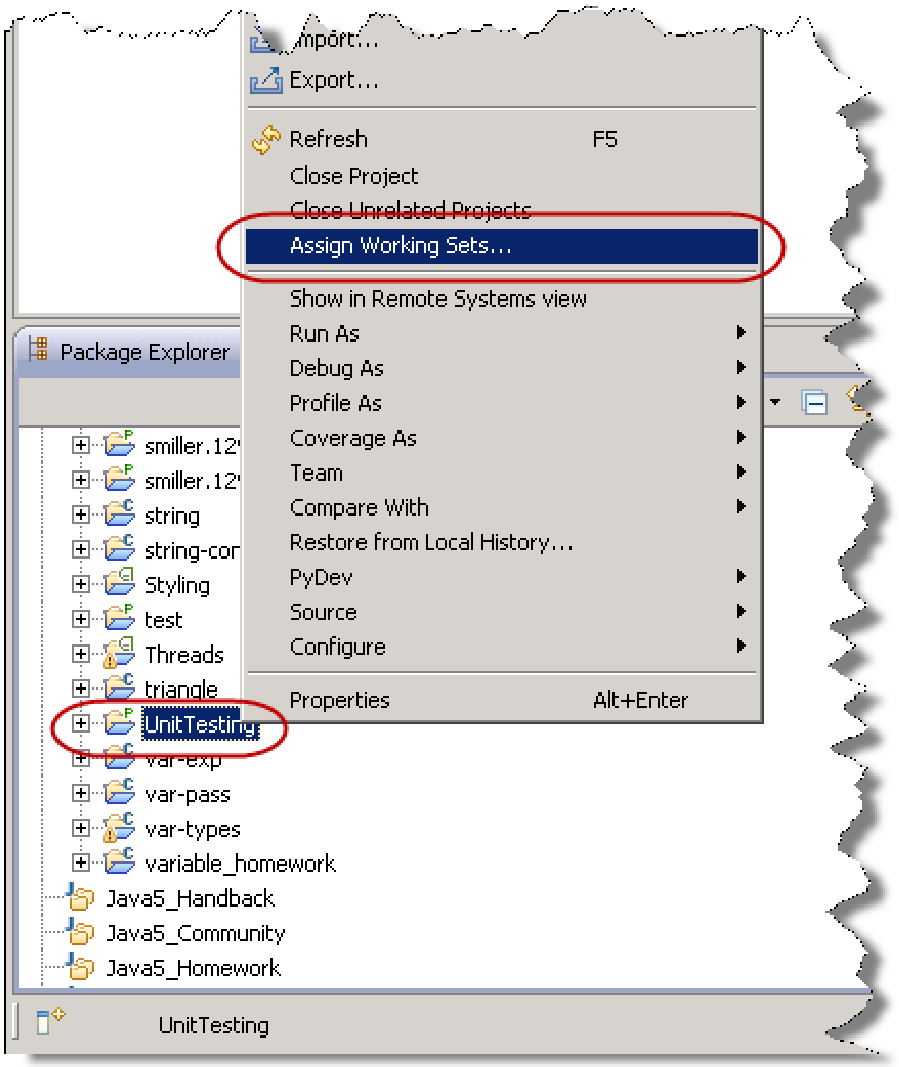
Click **OK** to return to the Pydev Project screen, then click **Finish**. You'll see a prompt to change perspectives. Check the **Remember my decision** box and then click **No**:


Open Associated Perspective?

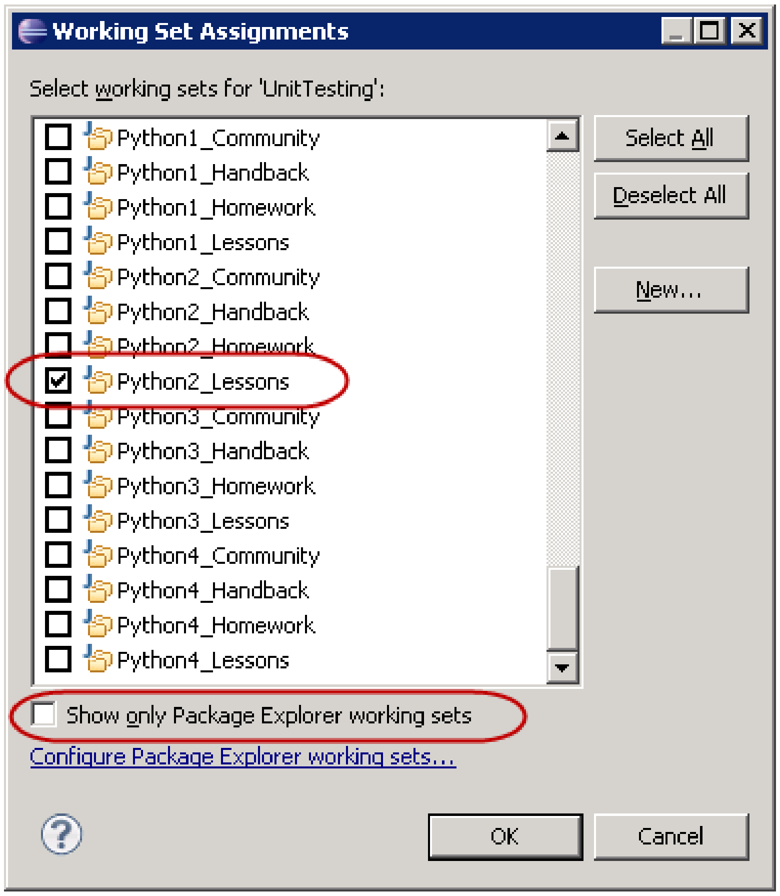
This kind of project is associated with the Pydev perspective. Do you want to open this perspective now?

- [x] Remember my decision

When you first create a PyDev project, it is placed in the Other Projects working set. It's a good idea to keep your Python projects together, so go ahead and put your newly created project into the Python2_Lessons working set. In the **Other Projects** working set, find the **UnitTesting** project. Right-click it and select **Assign Working Sets...**:

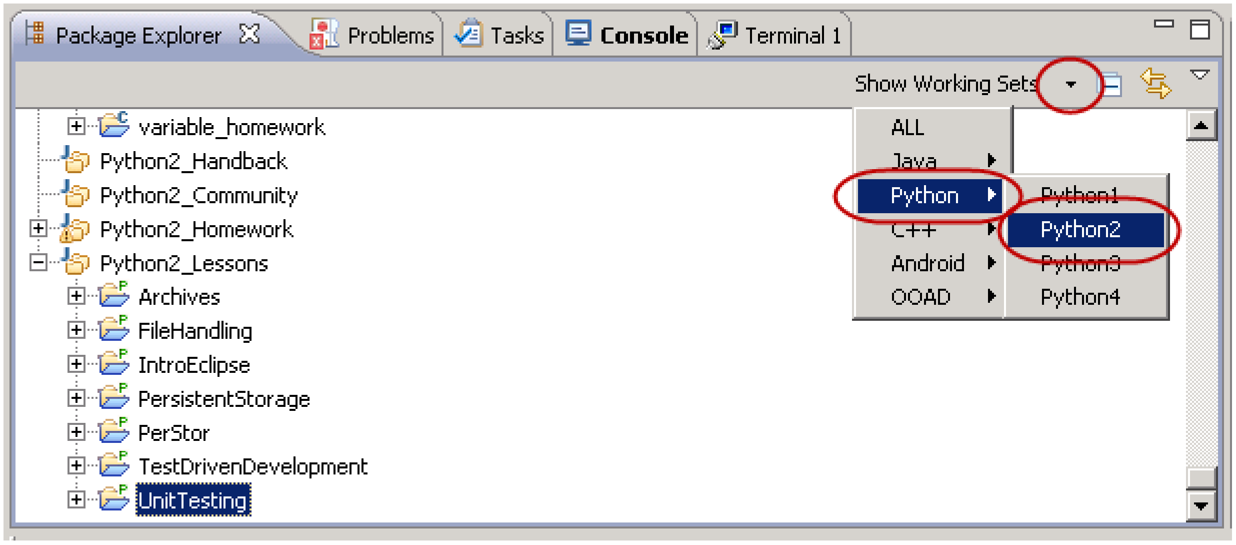


The **Working Set Assignments** screen appears. Click **Deselect All** to clear any selected working sets, and then check the box for the **Python2_Lessons** working set (the one for this course). Uncheck the **Show only Package Explorer working sets** box, and click **OK**:



You will need to do this for each new project you create. (Your working sets may differ from those shown here; you'll only see working sets for the courses in which you are enrolled.)

To see only the working sets for this course, click the drop-down arrow next to **Show Working Sets**, and select **Python | Python 2**:

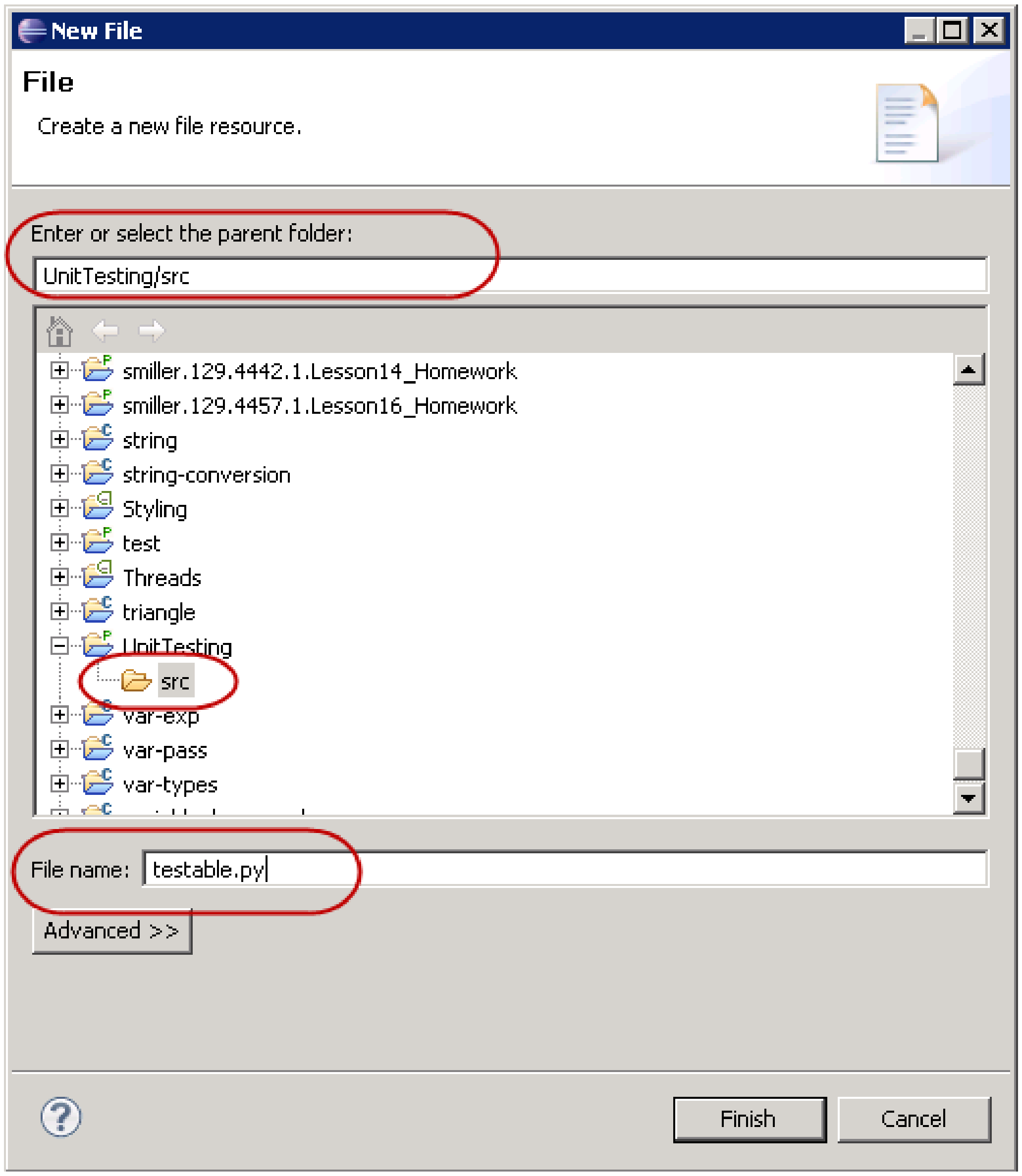


Your **UnitTesting** project is now listed in the **PyDev Package Explorer** panel on the lower left corner of your Eclipse screen, in your **Python2_Lessons** working set.

This hierarchical view of available resources (directories and files) in Eclipse is commonly called the *workspace*. You now have a *project* named **UnitTesting** in your workspace.

Before you go on, make sure that the **UnitTesting** project is displayed in the PyDev Package Explorer window. Click on this new project to select it.

From the **File** menu, select **New | File**. A New File dialog box appears. Select the **src** subdirectory of **UnitTesting**, enter the filename **testable.py**, and then click **Finish**:



Enter or select the parent folder:

UnitTesting/src

smiller.129.4442.1.Lesson14_Homework
smiller.129.4457.1.Lesson16_Homework
string
string-conversion
Styling
test
Threads
triangle
UnitTesting
src
var-exp
var-pass
var-types

File name: testable.py

A new editor window appears next to the workspace. We'll use this editor because it understands Python syntax. In **testable.py**, type the blue code as shown:

CODE TO TYPE:

```
"""Demonstrates the unittest module in action."""
import unittest

def cube(x):
    '''Returns the cube of a passed value'''
    return x*3

class TestCube(unittest.TestCase):

    def test_small_number(self):
        self.assertEqual(cube(3), 27, "Cube of 3 is not 27")

    def test_large_number(self):
        self.assertEqual(cube(1000), 1000000000, "Cube of 1000 should be 1000000
000")

    def test_bad_input(self):
        self.assertRaises(TypeError, cube, 'x')

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

To run the program, right-click in the editor window and select **Run As...**, and select **Python Run**. This program contains a bug: instead of returning its argument raised to the third power (cubed), the **cube()** function returns its argument multiplied by three. This is an easy mistake to make—we just omitted a single asterisk (*)—but it renders the function incorrect. When you run the program, you see output that looks something like this:

OBSERVE: Output from testable.py with an error in the cube() function

```
FFF
======================================================================
FAIL: test_bad_input (__main__.TestCube)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\UnitTesting\src\testable.py", line 17, in test_bad_input
    self.assertRaises(TypeError, cube, 'x')
AssertionError: TypeError not raised by cube

======================================================================
FAIL: test_large_number (__main__.TestCube)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\UnitTesting\src\testable.py", line 14, in test_large_number
    self.assertEqual(cube(1000), 1000000000, "Cube of 1000 should be 1000000000"
)
AssertionError: Cube of 1000 should be 1000000000

======================================================================
FAIL: test_small_number (__main__.TestCube)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\UnitTesting\src\testable.py", line 11, in test_small_number
    self.assertEqual(cube(3), 27, "Cube of 3 is not 27")
AssertionError: Cube of 3 is not 27

----------------------------------------------------------------------
Ran 3 tests in 0.032s

FAILED (failures=3)
```

Failures. Bummer. And not only do we have failures, our program gives us even more data than doctest did. For example, our program gives the number of tests, followed by the length of time it took to run the tests, and the tests themselves can be set up to pass messages to the person running the tests.

When you run the program, it calls the **unittest.main()** method, which runs the *unittest Test Runner*. The Test Runner looks in your code for test suites, which are identified as Classes that inherit from the **unittest.TestCase** class. These test suites contain a number of tests, which are class methods that begin with the word "test."

Because the assertions within your unittest methods raise AssertionErrors, the package reports them as test failures, and the output makes it clear that something is wrong with the program. In fact, because of the message arguments passed to the methods, you get a pretty good idea of what is going wrong. Now, fix the error by changing the operation in the **cube()** function to an exponentiation. Modify **testable.py** by adding the blue code as shown:

CODE TO TYPE:

```
"""Demonstrates the unittest module in action."""
import unittest

def cube(x):
    '''Returns the cube of a passed value'''
    return x**3

class TestCube(unittest.TestCase):

    def test_small_number(self):

        self.assertEqual(cube(3), 27, "Cube of 3 should be 27")

    def test_large_number(self):
        self.assertEqual(cube(1000), 1000000000, "Cube of 1000 should be 1000000
000")

    def test_bad_input(self):
        self.assertRaises(TypeError, cube, 'x')

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

Right-click in the editor window and select **Run As | Python unit-test**. With the error now corrected, your output from **testable.py** looks like this:

OBSERVE: Output from testing testable.py

```
Finding files... done.
Importing test modules ... done.

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Now, run it by right-clicking in the editor window and selecting **Run As | Python Run**:

OBSERVE: Output from running testable.py

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

OK
```

The three dots at the top represent the three tests. If they had failed, you would have seen an "F" replacing each failure. If there were significant errors, you would have seen an "E." Such error indications usually mean that something is wrong with your logic. You see the test count and the time for the duration of the tests' run. Chances are that for this basic test, you'll get a value of 0.000, but keep in mind that unittests are not performance tests. You'll cover performance tests in a later course.

Note We also corrected the failure on test_bad_input. Why? Because the string **x** can be "multiplied" to give **xxx**, so no TypeError is raised. Because exponentiation does not work with strings, the cube function must be fixed before the test passes.

Breaking Down Tests

Now that you have the tests working, consider how they work. Look over this color-coded test code:

OBSERVE:

```
"""Demonstrates the unittest module in action."""
import unittest

def cube(x):
'''Returns the cube of a passed value'''
   return x**3

class TestCube(unittest.TestCase):
    def test_small_number(self):
        self.assertEqual(cube(3), 27, "Cube of 3 should be 27")

    def test_large_number(self):
        self.assertEqual(cube(1000), 1000000000, "Cube of 1000 should be 1000000
000")

    def test_bad_input(self):
        self.assertRaises(TypeError, cube, 'x')
if __name__ == "__main__":
    unittest.main()
```

The **test_small_number()** method in the **TestCube** class has a single statement: a call to the **assertEqual()** method inherited from **unittest.testCase**. That statement contains an assertion that its first two arguments are equal—that **cube(3)** is equal to **27**. If the values do not match, then the assertion fails and the message **"Cube of 3 should be 27"** is returned during the test and reported by the framework.

If you include useful assertion error messages, they will help you remember what your tests are supposed to be doing. They will also help other programmers understand your tests. It's easier to figure out what to fix when error messages are meaningful (fortunately, the default messages produced by unittest have improved recently, as well).

In the third test, **test_bad_input()** checks to see if the **cube()** function throws a **TypeError** exception. The first argument provided is the expected exception; the second argument is the function to test; the remaining arguments will be passed to the function in question— the **cube()** function (a one-argument function, so you see a single additional argument **'x'**); it is possible to use both positional and keyword arguments (but the function you are testing doesn't take any keyword arguments). Using this method lets you verify that certain inputs raise specific exceptions.

Test-Driven Development: Tests As Specifications

Now that you've begun to appreciate the value of testing, follow the basic rule of test-driven development (TDD): only write code to make a failing test pass. This means that you begin your development projects by creating tests, which then act as a specification for the behavior of the program. By developing software this way, the programmer is forced to develop only the necessary functionality, and resists including extraneous elements. As the agile programming community says, "YAGNI"—You Ain't Gonna Need It. If it doesn't help you pass a test, it really isn't necessary.

Background of unittest

Kent Beck, the creator of *Extreme Programming* and *Test Driven Development*, wrote a testing framework for agile programming in the Smalltalk programming language. Later, along with Erich Gamma, he wrote a Java-based implementation of this test framework called JUnit. This test framework has since been ported to many other languages, including Python, where it is sometimes called "PyUnit."

The advantage of unittest is that the core concepts are tried and tested. This is important in a test framework because that means you can rely on it. As we learned in the previous course, if we refactor our code and it still passes the tests, we can be reasonably sure that we haven't introduced an error.

unittest uses these important concepts:

- **Test Fixtures:** The setup for your tests. Fixtures include creation of temporary databases, servers, and anything else needed to run the test. The fixtures frequently need to be cleaned up after a test. To use a spelling test analogy, think of a Test Fixture as a combination pencil, eraser, test sheet, and word list.
- **Test Cases:** Each test case is an individual test. It checks for a specific response to an assertion, and then is distilled to a boolean statement. Using the spelling test analogy again, think of a Test Case as a single question on the test.
- **Test Suite:** A test suite is a collection of Test Cases (or even other **Test Suites**). Returning to our spelling test again, think of a Test Suite as the set of all questions on the test sheet.
- **Test Runner:** The software that actually runs the tests. The runner can be launched from the command line, graphical interface, web interface, or any other input method. It returns special values to indicate the success of the tests, and these values can be evaluated by you or by various automated tools. In the spelling test analogy, the test runner would be you, the reader, going through the list of questions.

Comparing doctest and unittest

So, which should you use, doctest or unittest? To a certain extent, this is a matter of individual preference. Let's compare the two:

doctest	unittest
More readily accessible	More challenging to learn
Documents your code to some degree	Maintains a clean separation between tests and documentation
Harder to maintain as features change	Easier to maintain as features change
Assertions are more difficult to incorporate	Assertions are the primary tools for verifying correct performance
Verbose	Concise

The Python community generally agrees that while doctests have their place, unittests are usually more useful. doctests are easier to learn, but in the long run, unittests are the more streamlined choice. It is possible to integrate doctests in a unittest environment, though not quite as straightforward as you might like.

One Down

Congratulations! Just like that, you are now equipped with a second Python test framework. In the lessons to come, we'll use both test frameworks to check our work and build good programming habits. According to the tenets of agile programming, test-driven development is the way to go. TDD lets you continue to refactor your code without introducing errors, and it encourages other programmers to love you for your devotion to best practices. In the next lesson, we'll explore test-driven development even further. See you there!

Test-Driven Development

Agile Programming and Test-Driven Development

So far, we've learned that tests enable us to refactor code, and that refactoring lets us improve our code's clarity and performance. To support testing, we've learned two test frameworks in Python, *doctest* and *unittest*. With those tools in hand, we're ready to dive into *Test-Driven Development* (we'll call it TDD from now on) .

The concept of TDD is pretty straightforward. Once you've identified the requirements of a program, you begin creating it, not by coding, but by writing tests. After you're satisfied with the tests you've written (which may require lots of trial and error, but hey, you're human), you write the code that will pass the tests. The general outline for TDD workflow incorporates the mantra of agile programmers everywhere: "Do the simplest thing that could possibly work."

1. Write tests
2. Run tests
3. Write some code to pass the tests
4. Run tests
5. Refactor code
6. Repeat

And that's all there is to it.

You know, if you think about it, you've already done some TDD—well almost. In the projects for Python 1, as well as your first project for Python 2, you were given a set of requirements and then some expected results. In those cases, formal tests of your code which were performed by running the program, stimulating it with specific inputs, and observing and validating the results.

If you automate testing, you can repeat the tests reliably whenever you want. And thanks to doctest and unittest, you can include formal tests of your code in the lessons and projects to come.

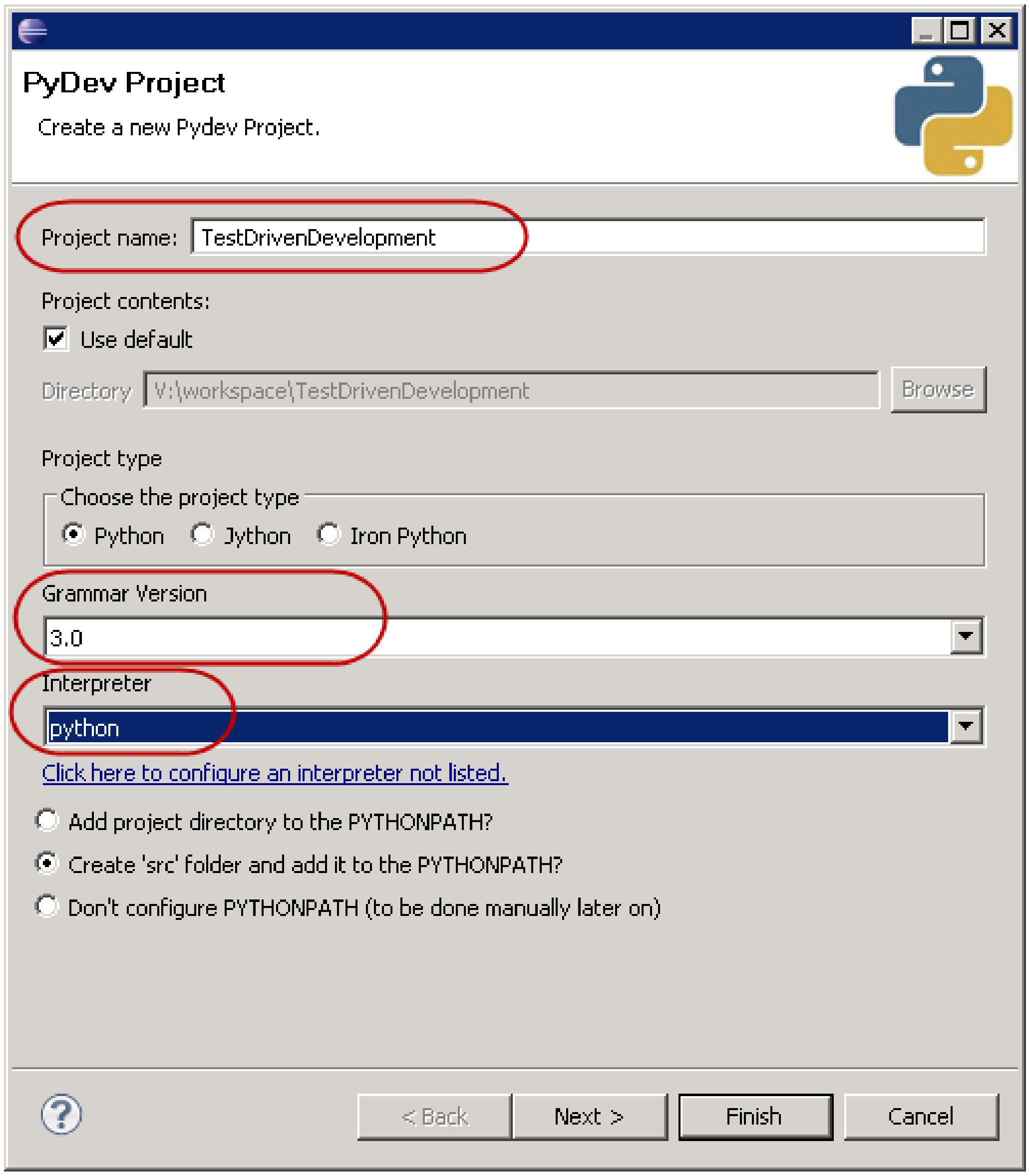
An Example of Test-Driven Development

Below is an example of the first step of TDD, **writing tests**. Suppose that you have been asked to develop an **adder(x, y)** function that takes two arguments and adds them together using a somewhat unusual definition of "add": integer + integer, string + string and list + list, use regular addition; integer + string converts the integer to a string before concatenation; and adding a string or an integer to a list, appends to the list (regardless of whether it's the first or second argument).

Create a **TestDrivenDevelopment** project and assign it to the **Python2_Lessons** working set. Then, create a source file named **testadder.py**. (If you remember how to do this, create it and go on to the section called editing and running. If you've forgotten the procedure for creating projects, assigning working sets, or creating source files, we'll give you detailed instructions one more time now.

Creating the Program

Select **File | New | PyDev Project** and create a **TestDrivenDevelopment** project as shown:



PyDev Project

Create a new Pydev Project.

Project name: TestDrivenDevelopment

Project contents:

[✓] Use default

Directory V:\workspace\TestDrivenDevelopment

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?

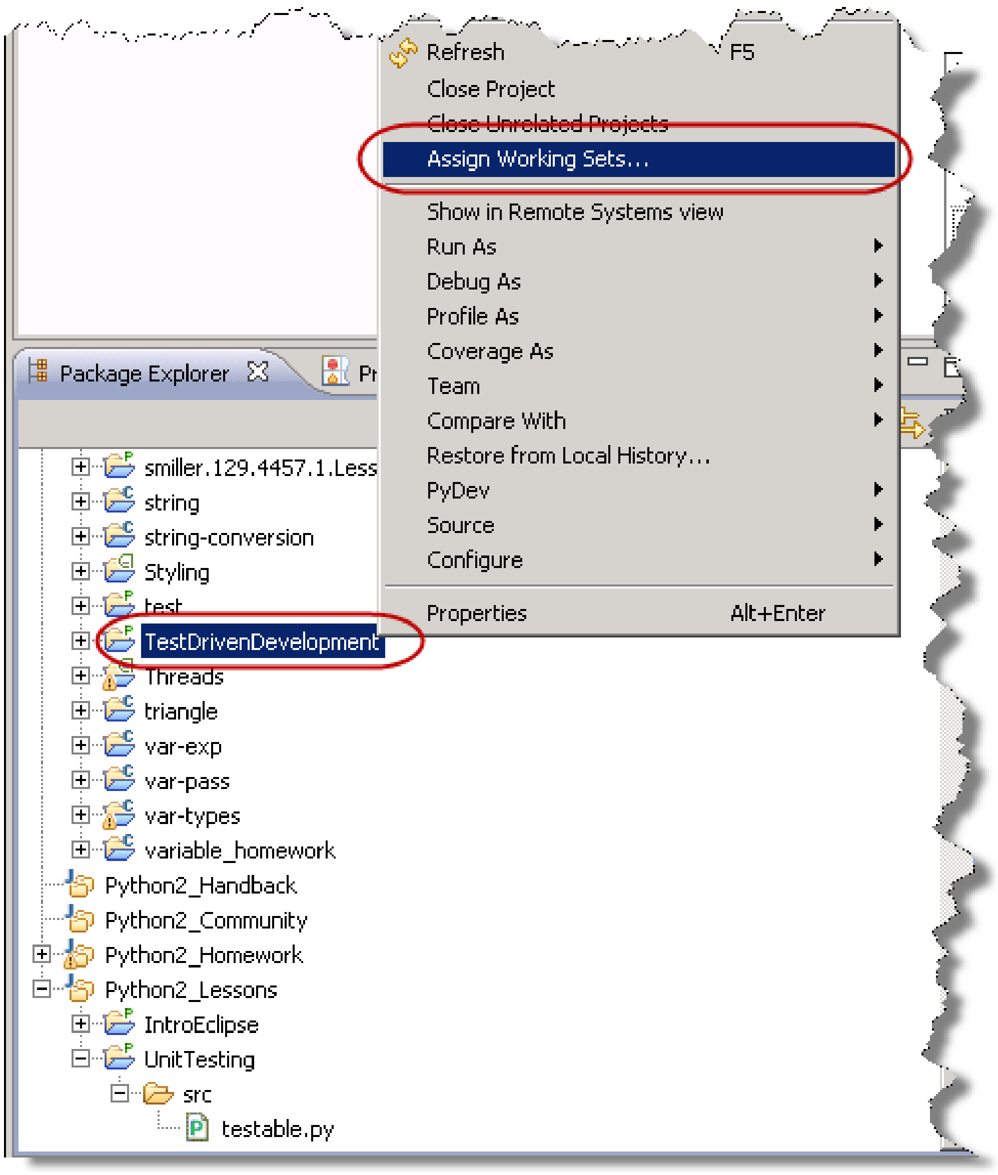
[● Create 'src' folder and add it to the PYTHONPATH?]

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

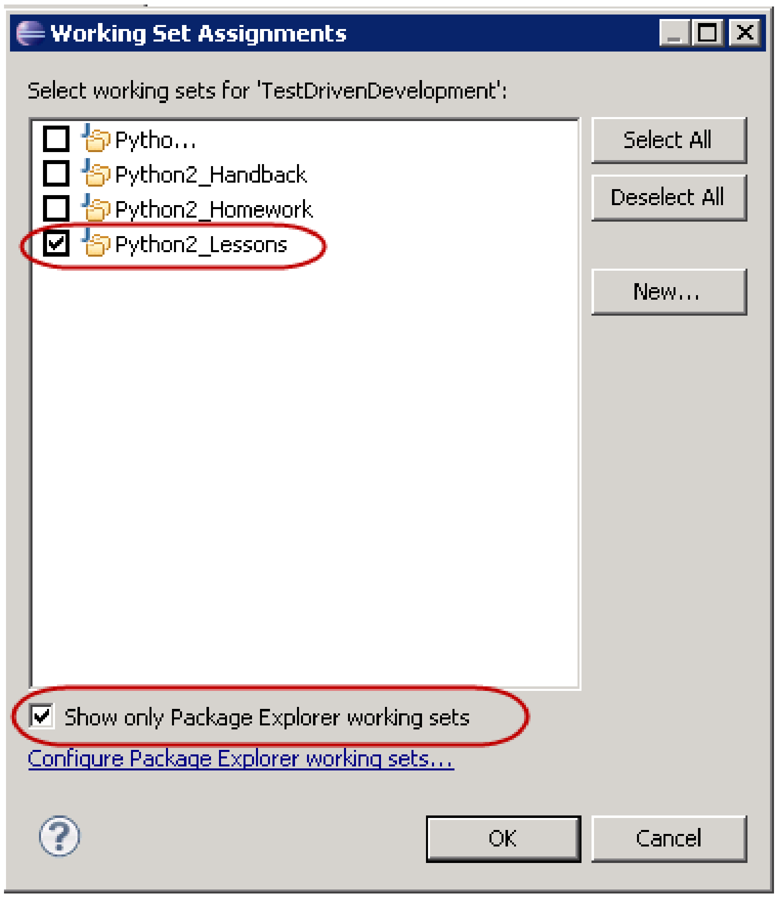
Click **Finish**.

If you're prompted to Open Associated Perspective, check the **Remember my decision** box and click **No**.

Your new project is located in the Other Projects working set in the Package Explorer. Find it, right-click it, and select **Assign Working Sets...**:



In the Working Set Assignments dialog, select **Show only Package Explorer working sets** and **Python2_Lessons**:


Working Set Assignments

Select working sets for 'TestDrivenDevelopment':

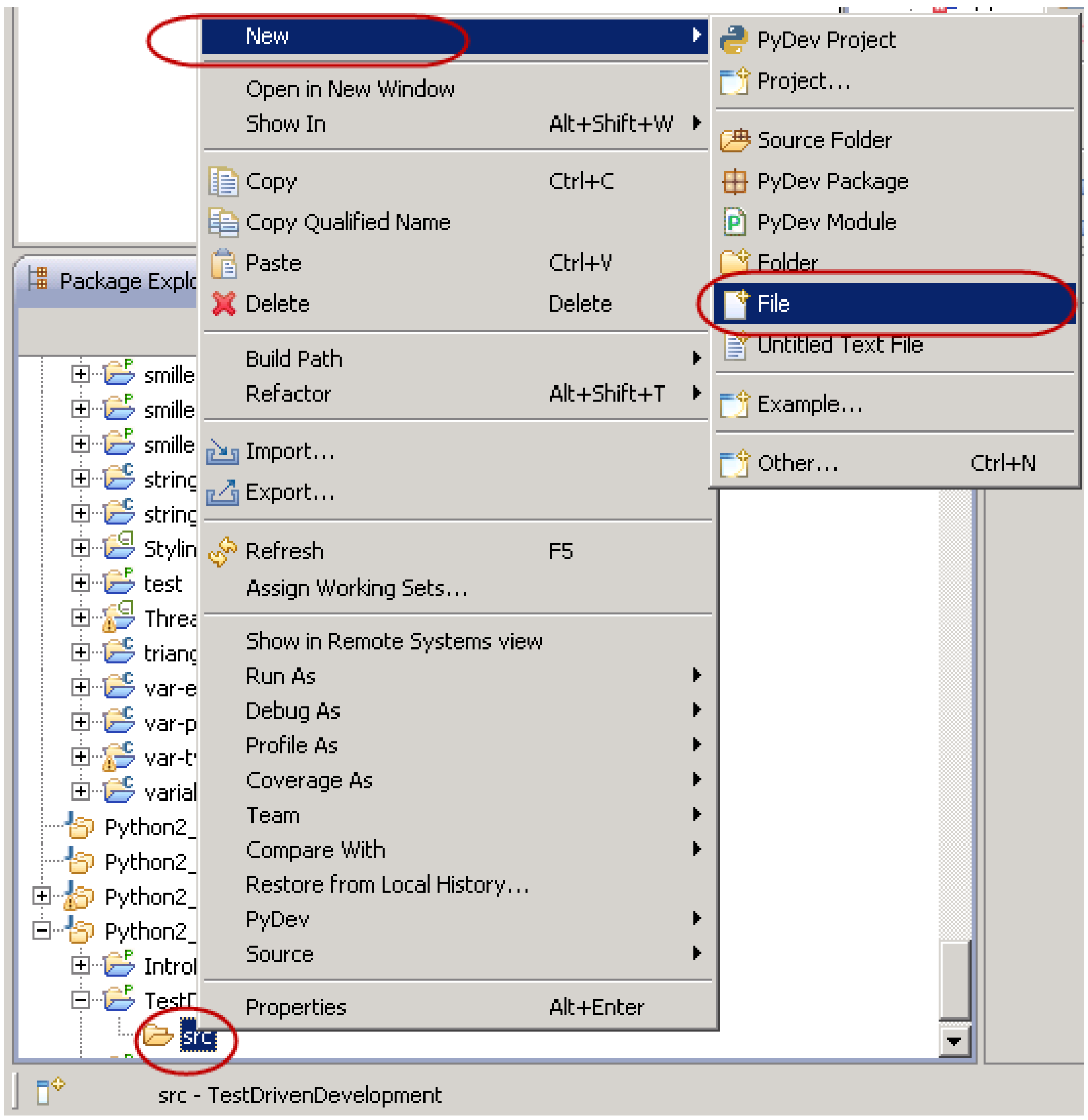
- [ ] Pytho...
- [ ] Python2_Handback
- [ ] Python2_Homework
- [✓] Python2_Lessons

[✓] Show only Package Explorer working sets

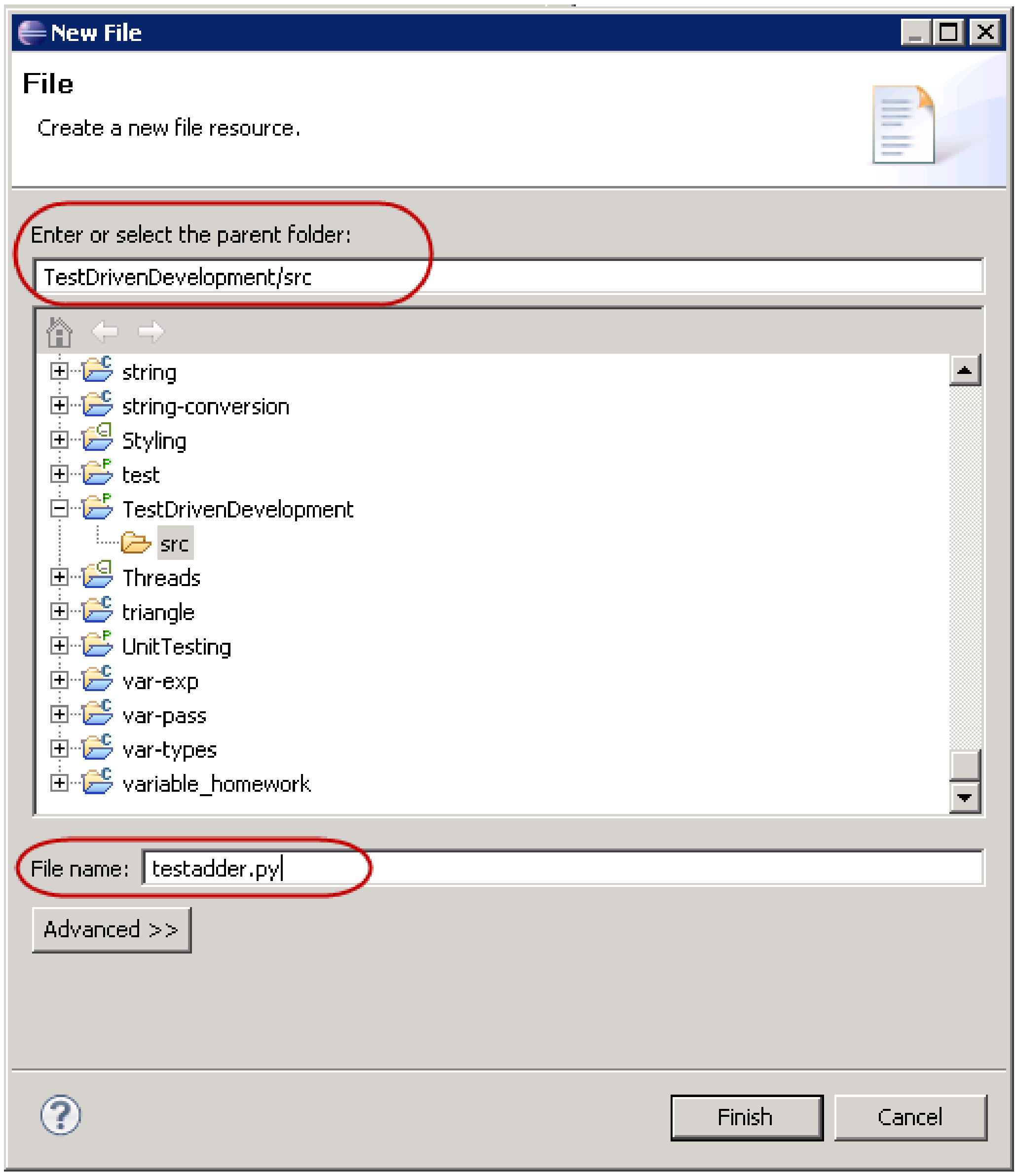
Configure Package Explorer working sets...

Click **OK**.

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



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


New File

File

Create a new file resource.

Enter or select the parent folder:

TestDrivenDevelopment/src

File name: testadder.py

The file now appears in the Eclipse editor window.

Editing and Running the Program

In **testadder.py**, type in the code below as shown:

CODE TO TYPE:

```
"""
Demonstrates the fundamentals of unittest.
adder() is a function that lets you 'add' integers, strings, and lists.
"""

from adder import adder # keep the tested code separate from the tests

import unittest
class TestAdder(unittest.TestCase):

    def test_numbers(self):
        self.assertEqual(adder(3,4), 7, "3 + 4 should be 7")

    def test_strings(self):
        self.assertEqual(adder('x','y'), 'xy', "x + y should be xy")

    def test_lists(self):
        self.assertEqual(adder([1,2],[3,4]), [1,2,3,4], "[1,2] + [3,4] should be
 [1,2,3,4]")

    def test_number_and_string(self):
        self.assertEqual(adder(1,'two'), '1two', "1 + two should be 1two")

    def test_numbers_and_list(self):
        self.assertEqual(adder(4,[1,2,3]), [1,2,3,4], "4 + [1,2,3] should be [1,
2,3,4]")

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

Don't run the program just yet. Although it imports an adder function (the function it's eventually going to test), that import will fail unless that function is defined. The simplest code we have to allow the test harness (automated test framework) to run, is an **adder** module that contains an empty **adder()** function. In your **TestDrivenDevelopment/src** folder, create **adder.py** as shown:

CODE TO TYPE:

```
"adder.py: defines an adder function according to a slightly unusual definition.
"

def adder(x, y):
    pass
```

Now let's go on to step two of the cycle, **run tests**.

Save the adder.py file.

Then, go back to **testadder.py** and run it.

OBSERVE: Output from testadder.py with an incomplete adder function

```
FFFFF
======================================================================
FAIL: test_lists (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 17, in test_l
ists
    self.assertEqual(adder([1,2],[3,4]), [1,2,3,4], "[1,2] + [3,4] should be [1,
2,3,4]")
AssertionError: [1,2] + [3,4] should be [1,2,3,4]

======================================================================
FAIL: test_number_and_string (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 20, in test_n
umber_and_string
    self.assertEqual(adder(1,'two'), '1two', "1 + two should be 1two")
AssertionError: 1 + two should be 1two

======================================================================
FAIL: test_numbers (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 11, in test_n
umbers
    self.assertEqual(adder(3,4), 7, "3 + 4 should be 7")
AssertionError: 3 + 4 should be 7

======================================================================
FAIL: test_numbers_and_list (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 23, in test_n
umbers_and_list
    self.assertEqual(adder(4,[1,2,3]), [1,2,3,4], "4 + [1,2,3] should be [1,2,3,
4]")
AssertionError: 4 + [1,2,3] should be [1,2,3,4]

======================================================================
FAIL: test_strings (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 14, in test_s
trings
    self.assertEqual(adder('x','y'), 'xy', "x + y should be xy")
AssertionError: x + y should be xy

----------------------------------------------------------------------
Ran 5 tests in 0.016s

FAILED (failures=5)
```

All five tests have failed. But we expected them to fail (yes, we did), because our **adder()** method doesn't actually do anything yet. While failed tests are not the ideal result, at least the tests didn't result in error messages. When you see error messages, they usually indicate the presence of a programming mistake, for instance, a function may have the wrong number of arguments, or a call to a method that an object doesn't have. But since our code didn't return any error messages, we can move on to step three of the TDD cycle: **write code to pass the tests**. In this first instance, we won't try and pass *all* of the tests, but instead provide a basic initial implementation that will pass *some* of them, then build from there. Edit **adder.py**, adding and ~~removing~~ code as shown:

CODE TO TYPE:

```
"adder.py: defines an adder function according to a slightly unusual definition.
"

def adder(x, y):
    ~~pass~~
    return x + y
```

Run **testadder** again.

OBSERVE: Ouput from the second run of testadder.py

```
.E.E.
======================================================================
ERROR: test_number_and_string (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 20, in test_n
umber_and_string
    self.assertEqual(adder(1,'two'), '1two', "1 + two should be 1two")
  File "V:\workspace\TestDrivenDevelopment\src\adder.py", line 4, in adder
    return x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'

======================================================================
ERROR: test_numbers_and_list (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 23, in test_n
umbers_and_list
    self.assertEqual(adder(4,[1,2,3]), [1,2,3,4], "4 + [1,2,3] should be [1,2,3,
4]")
  File "V:\workspace\TestDrivenDevelopment\src\adder.py", line 4, in adder
    return x + y
TypeError: unsupported operand type(s) for +: 'int' and 'list'

----------------------------------------------------------------------
Ran 5 tests in 0.016s

FAILED (errors=2)
```

The first line now contains three dots, each representing a successful test (give yourself a pat on the back for those!), and two "E" characters. Those E's represent errors that we get because our implementation works for only 60% of the test cases. That's not bad for a one-line function though, and the output from the test-run provides lots of information that helps us figure out how to stop the function from throwing exceptions and causing those errors.

The problems in our code seem to pop up when the arguments aren't of the same type. Since the function appears to do what we need it to do most of the time, we'll modify our program explicitly to change its performance just in the failing cases. We'll do that by adding an integer and a string, and adding an integer and a list (this last case should apply when adding anything to a list, not just an integer).

Edit your code as shown below:

CODE TO TYPE:

```
"adder.py: defines an adder function according to a slightly unusual definition.
"
import numbers

def adder(x, y):
    if isinstance(x, list):
        return x + [y]
    elif isinstance(y, list):
        return y + [x]
    elif isinstance(x, numbers.Number) and isinstance(y, str):
        return str(x) + y
    return x+y
```

We enhanced our code using the built-in **isinstance()** function. This function lets us check to see if a variable is of a particular type, or a subclass of that type. We have to import the **numbers** module in order to use **numbers.Number**, which is a superclass of all numeric types in Python.

Run **testadder** again. Now both of the original errors are fixed, but unfortunately, one of the test cases that succeeded previously is now broken. Don't worry too much—this a common occurrence. The good news is that the tests work and let us know about the problems!

OBSERVE: Results of the third testadder.py run

```
F....
======================================================================
FAIL: test_lists (__main__.TestAdder)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\testadder.py", line 17, in test_l
ists
    self.assertEqual(adder([1,2],[3,4]), [1,2,3,4], "[1,2] + [3,4] should be [1,
2,3,4]")
AssertionError: [1,2] + [3,4] should be [1,2,3,4]

----------------------------------------------------------------------
Ran 5 tests in 0.016s

FAILED (failures=1)
```

In the final version of our code, we want to make sure that the new special cases for lists are not applied when *both* arguments are lists. In those cases we want them to be left to the default **elif** case at the end of the function. Modify your code as shown:

CODE TO TYPE:

```
"adder.py: defines an adder function according to a slightly unusual definition.
"
import numbers

def adder(x, y):
    if isinstance(x, list) and not isinstance(y, list):
        return x + [y]
    elif isinstance(y, list) and not isinstance(x, list):
        return y + [x]
    elif isinstance(x, numbers.Number) and isinstance(y, str):
        return str(x)+y
    return x+y
```

Run **testadder** again. Nice. At last we have the pleasure of seeing all of our tests pass, with five dots on the first line of the output. Programmers who use unittest regularly often refer to themselves as "dot-addicted." It's amazing how gratifying it can be to see a row of dots printed out from a test!

OBSERVE: Output from testadder.py when all tests pass

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

More About the unittest.TestCase Class

The **TestCase** class is the cornerstone of the unittest module. We've learned to create our own test cases as subclasses of **TestCase**. Individual tests are written as methods of the subclass and have names that begin with the string "test." If you have only one test to run, you may implement that test as the class's **runTest()** method. You probably won't do that very much, but you may see it in other people's code, so it's worth knowing.

Test Fixture Set-up and Tear-down

If you want to define several tests, you could create a separate **TestCase** subclass for each one, but it's much simpler to create a single subclass with several test methods instead. So, why might you need more than one **TestCase** subclass? Well, one possibility is so that you can include **setUp()** and **tearDown()** methods, which would be run before and after each test method. In this case (as well as in others), grouping tests that require the same set-up and tear-down processing, is a good way to go.

Suppose you want to run some tests of code you have written that creates files. Each test needs to create files. And since the tests create random files (or at least since each test creates different files), if you run the tests in any old directory, clean-up could be difficult. To avoid creating such problems for ourselves, we'll write our code so that each test method creates the directory itself and cleans up the files it creates. To make our code even more efficient, we'll have it call a function to create the directory which was called by each test method. We could take it even further and create the directory within the **setUp()** method. This is called automatically before the framework calls each test method, just as the **tearDown()**method is called after each one. So we could use **tearDown()** to empty and delete the directory.

If the **setUp()** method raises an exception, the test framework will declare this test to have errors, and the test method will not be run. If it succeeds, the test is run, followed by the **tearDown()** method.

Let's check this out. In the **TestDrivenDevelopment/src** folder, create a new program named **setupDemo.py**. Type in the following code:

CODE TO TYPE:

```
"""
Demonstration of setUp and tearDown.
The tests do not actually test anything - this is a demo.
"""
import unittest
import tempfile
import shutil
import glob
import os

class FileTest(unittest.TestCase):

    def setUp(self):
        self.origdir = os.getcwd()
        self.dirname = tempfile.mkdtemp("testdir")
        print("Created", self.dirname)
        os.chdir(self.dirname)

    def test_1(self):
        "Verify creation of files is possible"
        for filename in ("this.txt", "that.txt", "the_other.txt"):
            f = open(filename, "w")
            f.write("Some text\n")
            f.close()
            self.assertTrue(f.closed)

    def test_2(self):
        "Verify that the current directory is empty"
        self.assertEqual(glob.glob("*"), [], "Directory not empty")

    def tearDown(self):
        os.chdir(self.origdir)
        shutil.rmtree(self.dirname)
        print("Deleted", self.dirname)

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

Here, you have defined a test case with two test methods. In order to make the test runnable anywhere, first the **setUp()** method saves the process's current directory (obtained with **os.getcwd()** in an instance variable). Then it uses **tempfile.mkdtemp()** to create a new temporary directory—the location it chooses will depend on your platform, so the method prints the directory path out for your inspection. Having created the new directory, **setUp()** then makes it the current directory.

The **tearDown()** method is called after each test. It makes the saved directory the current directory again (thereby ensuring that the temporary directory is no longer in use), and removes it (along with any content it may have) using **shutil.rmtree()**.

When you run the program, you might see something like this:

OBSERVE: Output from setupDemo.py

```
Created c:\docume~1\smiller\locals~1\temp\3\tmpm9h3hotestdir
Deleted c:\docume~1\smiller\locals~1\temp\3\tmpm9h3hotestdir
.Created c:\docume~1\smiller\locals~1\temp\3\tmpegpy6ltestdir
Deleted c:\docume~1\smiller\locals~1\temp\3\tmpegpy6ltestdir
.
----------------------------------------------------------------------
Ran 2 tests in 0.031s

OK
```

Here, the output from the test code itself is mixed with the **..** output from the testing framework, making it difficult to see exactly what's happening (though the absence of error messages is reassuring). It isn't usually a good idea to produce output from test cases for a couple of reasons. First, when the test succeeds there should be no output—this makes it much easier to determine whether tests have passed or failed. Second, it's quite possible that nobody will read that output anyway.

You may find that you prefer to run your tests using the features built-in to Ellipse to handle unit tests. To do so, select the **setupDemo.py** file and then choose **Run | Run As | Python unit-test**. Then, your output will look like this:

OBSERVE: Output from setupDemo.py

```
Finding files...
['V:\\workspace\\TestDrivenDevelopment\\src\\setupDemo.py'] ... done
Importing test modules ... done.

test_1 (setupDemo.FileTest)
Verify creation of files is possible ... Created c:\docume~1\smiller\locals~1\te
mp\3\tmpzo6uwatestdir
Deleted c:\docume~1\smiller\locals~1\temp\3\tmpzo6uwatestdir
ok
test_2 (setupDemo.FileTest)
Verify that the current directory is empty ... Created c:\docume~1\smiller\local
s~1\temp\3\tmpquthgmtestdir
Deleted c:\docume~1\smiller\locals~1\temp\3\tmpquthgmtestdir
ok

------------------------------------------------
Ran 2 tests in 0.031s

OK
```

The docstring for each test is now printed before the test starts, but the **print** statements are definitely interfering with the output. One way to correct this would be to remove the top-level instructions that call the **unittest.main()** function. When you ask Ellipse to run the program as a unit test, it automatically performs the work in the top-level instructions anyway. But in most cases, you'll want to retain that code; without it the program will not run correctly as a stand-alone module (run from outside Ellipse).

So instead, we'll remove the **print** statements when we modify our code. Let's do that now. Edit **setupDemo.py** as shown:

CODE TO TYPE:

```
"""
Demonstration of setUp/tearDown.
The tests do not actually test anything much - this is a demo.
"""
import unittest
import tempfile
import shutil
import glob
import os

class FileTest(unittest.TestCase):

    def setUp(self):
        self.origdir = os.getcwd()
        self.dirname = tempfile.mkdtemp("testdir")
        ~~print("Created", self.dirname)~~
        os.chdir(self.dirname)

    def test_1(self):
        "Verify creation of files is possible"
        for filename in ("this.txt", "that.txt", "the_other.txt"):
            f = open(filename, "w")
            f.write("Some text\n")
            f.close()
            self.assertTrue(f.closed)

    def test_2(self):
        "Verify that the current directory is empty"
        self.assertEqual(glob.glob("*"), [], "Directory not empty")

    def tearDown(self):
        os.chdir(self.origdir)
        shutil.rmtree(self.dirname)
        ~~print("Deleted", self.dirname)~~

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

Run this module using **Run | Run As | Python Run**; your output looks like this:

OBSERVE: Output from setupDemo.py

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.031s

OK
```

Test Case Enumeration

When you run **unittest.main()**, or when Eclipse runs a program as a Python unit test, all subclasses of **unittest.TestCase** are taken from the module. An instance of each subclass is created, and each method of the class with a name that begins with "test" is called. (These calls are preceded by a call to the **setUp()** method if it exists, and followed by a call to the **tearDown()** method if it exists).

All of the above actions are taken when we call the TestCase's **run()** method. The **TestCase** class records the results of the call in a special object, and they are summarized in the output of the test framework, after all tests have been run.

TestCase Methods

There are a number of methods you can call to make assertions about your program's state. The most commonly used TestCase Methods are:

TestCase Method	Description
assertTrue(expr[, msg])	Unless **expr** evaluates as true, the test fails.
assertFalse(expr[, msg])	If **expr** evaluates as true, the test fails.
assertEqual(first, second[, msg])	Unless **first** and **second** are equal, the test fails.
assertNotEqual(first, second[, msg])	If **first** and **second** are equal, the test fails.
assertAlmostEqual(first, second[, places[, msg]])	Computes the difference between **first** and **second** and rounds it to **places** decimal places. If the rounded result is non-zero, the test fails.
assertNotAlmostEqual(first, second[, places[, msg]])	Computes the difference between **first** and **second** and rounds it to **places** decimal places. If the rounded result is zero, the test fails.
assertRaises(exception, callable, ...)	Calls **callable**, passing it any positional and keyword arguments that follow. If the call does not raise the given exception, the test fails.

The methods above do have alternative names (**assertTrue()**, for example, is also known as **assert_()**), but the names above are preferred. Most of these methods take an optional message argument. If you don't provide a message, unittest will try to formulate one that gives you as much information as possible. To test this, create a new program named **messagetest.py** in your **TestDrivenDevelopment/src** project folder as shown:

CODE TO TYPE:

```
"""
Demonstrate a message formulated by the unittest system.
"""

import unittest

class DemoCase(unittest.TestCase):
    def testMessage1(self):
        self.assertEqual([1,2,3,4], [1, 2, [3, 4]])

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

Run it using **Run | Run As | Python Run**; the output looks something like this:

OBSERVE: Output from messagetest.py

```
F
======================================================================
FAIL: testMessage1 (__main__.DemoCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\TestDrivenDevelopment\src\messagetest.py", line 9, in testM
essage1
    self.assertEqual([1,2,3,4], [1, 2, [3, 4]])
AssertionError: Lists differ: [1, 2, 3, 4] != [1, 2, [3, 4]]

First differing element 2:
3
[3, 4]

First list contains 1 additional elements.
First extra element 3:
4

- [1, 2, 3, 4]
+ [1, 2, [3, 4]]
?        +     +



----------------------------------------------------------------------
Ran 1 test in 0.016s

FAILED (failures=1)
```

The system has performed a fairly detailed analysis of the differences between the two lists, and points out, in the lengthy message, that the lists differ at element 2, and that the first list has an extra element. This informative message is the result of some recent clean-up work that was done to Python's unittest module. With this tool available, now if you can't come up with a particularly good error message yourself, you can try letting the system generate one for you.

Laying the Foundation

In this lesson, you've learned about some basic functions of the unittest module. This will serve you well during the course, but we've only scratched the surface of unittest!

You have also learned to engage test-driven development practices. For the rest of this course, and all following courses in the OST Python series, you'll be required to use this methodology, in fact, your instructor will verify that you've solved problems successfully by running tests against it. By the end of this course, you'll be really comfortable with TDD and unittest, and writing tests will become second nature to you!

In the next lesson, we'll learn about some of Python's file-handling abilities. Keep up the excellent work and see you in a bit!

File Handling

High-Level File Operations

Now that we have a framework for testing and developing our code, it's time to start looking at some of Python's other built-in modules. In the next few lessons, we'll learn about various Python features, and we'll use *TDD* to develop small programs with the new features that we learn.

In this lesson, we will explore some of Python's high-level file handling capabilities. Python has lots of built-in functions and modules geared to help streamline the file handling process. It smooths over many differences between operating system platforms, so you'll have a single interface for dealing with files, whether you're on Windows, OS X, or Linux. First we'll review how to read and write files, then, we'll learn how to get information from and navigate in our file system, search for files, and archive and compress our files. We'll be playing with these features:

- the file object and the built-in open() function
- os.path
- glob

The File Object and the Built-in open() Function

Our first example involves the file object. You will create a module that can *read in*the contents of a file as a list of lines (without using file.readline or file.readlines), and *write out* a list of lines as a file. When the **read()** function is applied to the file that **write()** creates, it produces the same list as that which is passed in to the **write()** function. Unlike standard file methods, these functions deal with lines that do not contain the terminating newline.

You'll use newline as the delimiter. The file that you get after you write out a list containing the delimiter, does not need to produce the same list when it's read back in, so you don't have to figure out whether the lines contain the delimiter.

The **setUp()** method establishes a common file name and creates a set of test fixtures (particular lists that we have arbitrarily chosen to test the code). Each of the individual test methods calls a common **verify_file()** function with one of the test fixtures as its second argument.

Let's start by writing some tests, **test_fileops.py**, and stubbing out (that is, creating a "stub" program that doesn't do anything, so the other program(s) calling it don't show errors) your module, **fileops.py**. Don't forget to add a new test case if you add a new fixture!

Create your **FileHandling** project and assign it to the **Python2_Lessons** working set. Then create **test_fileops.py** as shown:

CODE TO TYPE:

```
import unittest
import os
import fileops

class TestReadWriteFile(unittest.TestCase):
    """Test case to verify list read/write functionality."""

    def setUp(self):
        """This function is run before each test."""
        self.fixture_file = r"v:\workspace\FileHandling\src\test-read-write.txt"
        self.fixture_list = ["my", "written", "text"]
        self.fixture_list_empty_strings = ["my", "", "", "written", "text"]
        self.fixture_list_trailing_empty_strings = ["my", "written", "text", "", ""]

    def verify_file(self, fixture_list):
        """Verifies that a given list, when written to a file,
        is returned by reading the same file."""
        fileops.write_list(self.fixture_file, fixture_list)
        observed = fileops.read_list(self.fixture_file)
        self.assertEqual(observed, fixture_list,
                         "%s does not equal %s" % (observed, fixture_list))

    def test_read_write_list(self):
        self.verify_file(self.fixture_list)

    def test_read_write_list_empty_strings(self):
        self.verify_file(self.fixture_list_empty_strings)

    def test_read_write_list_trailing_empty_strings(self):
        self.verify_file(self.fixture_list_trailing_empty_strings)

    def tearDown(self):
        """This function is run after each test."""
        try:
            os.remove(self.fixture_file)
        except OSError:
            pass

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

Generally, each unit test should test just one function or method at a time. Otherwise our code will produce fragile tests, that may break as code is refactored. Our example is a special case, though. We're trying to match the input of **write_list()** with the output of **read_list()**, and rewriting the implementation of one function in our tests just to test the other seems redundant.

You'll see an error marker on the **import fileops** line because we haven't created **fileops.py** yet, so we can't run this program.

Now, let's *stub* the functions in **fileops.py**. The stubbed module provides functions with the correct interface, but no functionality. We don't expect the tests to succeed when we run them, but if the stubbed module is correctly structured we'll see *failures* rather than *errors*.

CODE TO TYPE:

```
"""Reads a list from a file and writes a list to a file."""

def write_list(fn, lst):
    """Writes a list to a named file. Each list item will be on
    a separate line. Overwrites the file if it already exists.
    """
    pass

def read_list(fn, lst):
    """Reads a list from a file without using readline.
    Uses standard line endings ("\n") to delimit list items.
    """
    pass
```

Save **fileops.py**, then run **test_fileops.py**. Your output will look like this:

OBSERVE: Output from running test_fileops.py with the stubbed module

```
EEE
======================================================================
ERROR: test_read_write_list (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 24, in test_read_write_lis
t
    self.verify_file(self.fixture_list)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 19, in verify_file
    observed = fileops.read_list(self.fixture_file)
TypeError: read_list() takes exactly 2 positional arguments (1 given)

======================================================================
ERROR: test_read_write_list_empty_strings (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 27, in test_read_write_lis
t_empty_strings
    self.verify_file(self.fixture_list_empty_strings)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 19, in verify_file
    observed = fileops.read_list(self.fixture_file)
TypeError: read_list() takes exactly 2 positional arguments (1 given)

======================================================================
ERROR: test_read_write_list_trailing_empty_strings (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 30, in test_read_write_lis
t_trailing_empty_strings
    self.verify_file(self.fixture_list_trailing_empty_strings)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 19, in verify_file
    observed = fileops.read_list(self.fixture_file)
TypeError: read_list() takes exactly 2 positional arguments (1 given)

----------------------------------------------------------------------
Ran 3 tests in 0.016s

FAILED (errors=3)
```

The "E" reports indicate that there is some mismatch between the tests and the stub. You need to get rid of any such problems before you replace the stubs with real functionality. The error messages let us know that we're expecting too many arguments in our **read_list()** function. Modify **fileops.py** as shown:

CODE TO TYPE:

```
"""Reads a list from a file and writes a list to a file."""

def write_list(fn, lst):
    """Writes a list to a named file. Each list item will be on
    a separate line. Overwrites the file if it already exists.
    """
    pass

def read_list(fn~~, lst~~):
    """Reads a list from a file without using readline.
    Uses standard line endings ("\n") to delimit list items.
    """
    pass
```

Save it, and then run **test_fileops.py**. All the tests fail with "F" now, but that's a good thing—it means that the interfaces in the tests match those in the stubbed code. Your output will look something like this:

OBSERVE: Output from running test_fileops.py with the stubbed module

```
FFF
======================================================================
FAIL: test_read_write_list (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 24, in test_read_write_lis
t
    self.verify_file(self.fixture_list)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 21, in verify_file
    "%s does not equal %s" % (observed, fixture_list))
AssertionError: None does not equal ['my', 'written', 'text']

======================================================================
FAIL: test_read_write_list_empty_strings (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 27, in test_read_write_lis
t_empty_strings
    self.verify_file(self.fixture_list_empty_strings)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 21, in verify_file
    "%s does not equal %s" % (observed, fixture_list))
AssertionError: None does not equal ['my', '', '', 'written', 'text']

======================================================================
FAIL: test_read_write_list_trailing_empty_strings (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 30, in test_read_write_lis
t_trailing_empty_strings
    self.verify_file(self.fixture_list_trailing_empty_strings)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 21, in verify_file
    "%s does not equal %s" % (observed, fixture_list))
AssertionError: None does not equal ['my', 'written', 'text', '', '']

----------------------------------------------------------------------
Ran 3 tests in 0.016s

FAILED (failures=3)
```

The **FAIL** messages include enough traceback to identify the specific lines that are causing problems in the tests, and the error messages give you a pretty clear idea of what needs to be fixed (hint: don't return **"None"**!)

So now, let's fill out the stubs with real code. Modify **fileops.py** as shown:

CODE TO TYPE:

```
"""Reads a list from a file and writes a list to a file."""

def write_list(fn, lst):
    """Writes a list to a file. Each list item will be on a separate line.
    Overwrites the file if it already exists."""
    f = open(fn, "w")
    for item in lst:
        f.write("%s\n" % item)
    f.close()
    ~~pass~~

def read_list(fn):
    """Reads a list from a file without using readline. Uses unix style line
    endings ("\n") to delimit list items."""
    f = open(fn, "r")
    s = f.read()
    l = s.split("\n")
    return l
    ~~pass~~
```

This looks like it might work, so let's run our tests again. Bummer—more failures. Can you work out what the problem is, using the information in the messages?

OBSERVE: Output from test_fileops.py after filling out the stub code in fileops.py

```
FFF
======================================================================
FAIL: test_read_write_list (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 24, in test_read_write_lis
t
    self.verify_file(self.fixture_list)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 21, in verify_file
    "%s does not equal %s" % (observed, fixture_list))
AssertionError: ['my', 'written', 'text', ''] does not equal ['my', 'written', 'text']

======================================================================
FAIL: test_read_write_list_empty_strings (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 27, in test_read_write_lis
t_empty_strings
    self.verify_file(self.fixture_list_empty_strings)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 21, in verify_file
    "%s does not equal %s" % (observed, fixture_list))
AssertionError: ['my', '', '', 'written', 'text', ''] does not equal ['my', '', '', 'wr
itten', 'text']

======================================================================
FAIL: test_read_write_list_trailing_empty_strings (__main__.TestReadWriteFile)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_fileops.py", line 30, in test_read_write_lis
t_trailing_empty_strings
    self.verify_file(self.fixture_list_trailing_empty_strings)
  File "V:\workspace\FileHandling\src\test_fileops.py", line 21, in verify_file
    "%s does not equal %s" % (observed, fixture_list))
AssertionError: ['my', 'written', 'text', '', '', ''] does not equal ['my', 'written',
'text', '', '']

----------------------------------------------------------------------
Ran 3 tests in 0.047s

FAILED (failures=3)
```

If you examine the results carefully, you'll see that each observed result from the **read_line()** function contains an extra empty string. The problem is that your **write_list()** function is inserting a **newline** after each line it writes. When you read the file back in with the **read_list()** function, the **split("\n")** method expects strings on either side of each delimiter, so an extra blank line appears.

We can write our code to anticipate those newlines, but we have to make sure that we our files are still handled correctly in other ways. It's possible for a file, under certain circumstances, to be written *without* a final newline. The fix should take that possibility into account and take action only when the final character in the file is a newline terminator. Apply the fix as shown:

CODE TO TYPE:

```
"""Reads a list from a file and writes a list to a file."""

def write_list(fn, lst):
    """Writes a list to a file. Each list item will be on a separate line.
    Overwrites the file if it already exists."""
    f = open(fn, "w")
    for item in lst:
        f.write("%s\n" % item)
    f.close()

def read_list(fn):
    """Reads a list from a file without using readline. Uses unix style line
    endings ("\n") to delimit list items."""
    f = open(fn, "r")
    s = f.read()
    # If the last character in the file is a newline, delete it
    if s[-1] == "\n":
        s = s[:-1]
    l = s.split("\n")
    return l
```

Run it again. Ah. Success! We finally see the correct result:

OBSERVE:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.109s

OK
```

Good job.

Retrieving File and Path Name Information with os.path

The file system identifies files by name and location. The technical term for the name-and-location data is a *path* or *path name*. It details how to navigate through a sequence of folders to the required file. You can extract information from these path names by using the **os.path** module. Different platforms have different path name conventions (for example, Windows uses "\" as its path name separator while Unix-like operating systems use "/").

os.path is actually just a reference to another module that is platform specific. When your system loads the **os** module, code in that module selects and loads the appropriate submodule as **os.path**. On Windows, the submodule being used behind the scenes is **os.ntpath**. It has the same interface as os.path, so you can use most functions interchangeably. But using os.ntpath on its own means that you can only use Windows-style path names. **os.posixpath** is the path module for all operating systems that use Unix-style path names, such as Linux and OS X.

os.path contains utility functions for retrieving path name and file attribute information. Open an interactive session to see what **os.path** can do. We'll start out by creating a **temp directory** using its **mkdir()** function, and then go ahead and use other features. In an interactive shell, type the code as shown:

INTERACTIVE SESSION:

```
>>> import os
>>> os.mkdir(r"v:\tmp")
>>> f1 = open(r"v:\tmp\file1.txt", "w")
>>> f2 = open(r"v:\tmp\file2.txt", "w")
>>> f1.close()
>>> f2.close()
>>> f1.name
'v:\\tmp\\file1.txt'
>>> f2.name
'v:\\tmp\\file2.txt'
>>> os.path.exists(f1.name)
True
>>> os.path.exists(f2.name)
True
>>> os.path.exists(r"v:\tmp\file3.txt")
False
>>> os.path.getmtime(f1.name)
1270492734.5412514
>>> os.path.getmtime(f2.name)
1270492746.6686897
>>> os.path.basename(f1.name)
'file1.txt'
>>> os.path.basename("v:\\tmp\\")
''
>>> name, extension = os.path.splitext(f1.name)
>>> name
'v:\\tmp\\file1'
>>> extension
'.txt'
>>> os.path.dirname(f1.name)
'v:\\tmp'
>>> os.path.split(f1.name)
('v:\\tmp', 'file1.txt')
>>> joined = os.path.join(r"v:\tmp", "file1.txt")
>>> joined
'v:\\tmp\\file1.txt'
>>> os.path.exists(joined)
True
>>> joined = os.path.join(os.path.dirname(f1.name), os.path.basename(f1.name))
>>> joined
'v:\\tmp\\file1.txt'
>>> os.path.abspath(r"v:\tmp\..\tmp\file1.txt")
'v:\\tmp\\file1.txt'
```

os.path.exists() returns **True** if the path passed as an argument actually exists. On some platforms, the return value may differ based on file permissions and symbolic links.

os.path.getmtime() returns the amount of time (in seconds) between your platform's epoch date (the origin of time for your particular platform—for example, for Windows, getmtime would return the number of seconds since January 1st, 1601) and the last time that a file was modified. **getmtime()** is part of a group of functions that retrieves time information from a file. **getatime()** returns the last time the file was accessed and **getctime()** returns the time the file was created (on Unix-like systems, this is actually the last time a file was changed). You can convert these times to human-readable strings using functions from the **time** module, which we will look at later in this course.

As the module's name implies, **os.path** contains functions for manipulating path names. **os.path.basename()** returns the last path name component without any slashes. You can consider the basename as you would an actual filename component of a full path. If the path supplied to **basename()** ends in a slash, an empty string will be returned (because there is no filename component). To retrieve the path to the file, but not the file name itself, you can use **os.path.dirname()**.

Note In the os.path.basename example, we can't create a raw string literal ending with a single backslash (r"v:\tmp\"), so we instead used the non-raw string with double backslashes ("v:\\tmp\\"). Although backslashes are mostly treated as normal characters in raw string literals rather than altering the significance of the following character, any following quote character is always treated as part of the string literal. This is so that quote characters can still appear in string literals. For more information, see this stackoverflow article.

The **os.path.split()** function returns a tuple. The tuple's first element is what **dirname()** would return; its second element is what **basename()** would return.

os.path.join() does the opposite of **split()**; it joins path components together into full path names. It will add a slash between components where necessary, and you can give it as many arguments as you like. Joining the **dirname()** and **basename()** of a path gives back the original path.

Finding Path Names Using glob

So now you know how to read and write files, but what if you want to find a file? For that, you'll need the **glob()** function, which lives in the module of the same name. **glob()** finds paths that match a particular pattern. The symbols and patterns in the table below are the same *wildcards* you might use in your command shell and many other places:

Symbol	Description	Example
?	Match any single character exactly once.	?ar matches bar or tar, but not star.
*	Match any number of characters.	*ar matches bar, tar, star and exemplar.
[characters or character range]	Match exactly one character in a range or set.	[a-z]ar matches tar, but not star or 4ar

Now, using the interactive shell, we're going to create a directory containing the following files: **test1.txt**, **test2.txt**, **test3.txt**, and **another.one**. Let's see what **glob()** can do with these files. Type this code into an interactive Python console:

INTERACTIVE SESSION:

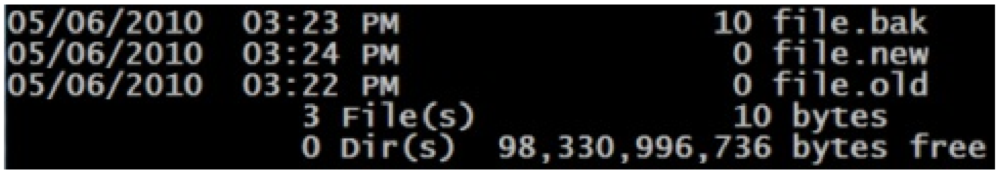
```
>>> for i in range(1,4):
...     f  = open(r"v:\tmp\test"+str(i)+".txt", "w")
...     f.close()
...
>>> f = open(r"v:\tmp\another.one", "w")
>>> f.close()
>>> import glob
>>> os.chdir(r"v:\tmp")
>>> glob.glob("*.*")
['another.one', 'test1.txt', 'test2.txt', 'test3.txt']
>>> glob.glob("*.txt")
['test1.txt', 'test2.txt', 'test3.txt']
>>> glob.glob("*.one")
['another.one']
>>> glob.glob("test?.txt")
['test1.txt', 'test2.txt', 'test3.txt']
>>> glob.glob("test[1-2].txt")
['test1.txt', 'test2.txt']
```

As long as their names share a common pattern, you can access your chosen files. There are also ways to read all of the entries within a directory, or even to walk through an entire directory tree, but we'll address that in a later course.

An Application to Sort and Retrieve File Information

Let's try using the **glob** and **os.path** modules to create a function that returns a list of the most recently modified files from a particular path. It will take as arguments, the number of files that we want returned, and the path where we'll look for the files. You'll reuse and modify the module from our last example, so don't worry about error handling just yet. To develop the good programming habits you're going to have, start out with some tests!

In the directory listing below, file.old is the oldest of the three listed files, and file.new the newest:



```
05/06/2010  03:23 PM                10 file.bak
05/06/2010  03:24 PM                 0 file.new
05/06/2010  03:22 PM                 0 file.old
               3 File(s)            10 bytes
               0 Dir(s)  98,330,996,736 bytes free
```

In your **FileHandling** project, create a new file named **test_latest.py** as shown:

CODE TO TYPE:

```
import unittest
import latest
import time
import os

PATHSTEM = "v:\\workspace\\FileHandling\\src\\"

class TestLatest(unittest.TestCase):

    def setUp(self):
        self.path = PATHSTEM
        self.file_names = ["file.old", "file.bak", "file.new"]
        for fn in self.file_names:
            f = open(self.path+fn, "w")
            f.close()
            time.sleep(1)

    def test_latest_no_number(self):
        """
        Ensure that calling the function with no arguments returns
        the single most recently-created file.
        """
        expected = [self.path + "file.new"]
        latest_file = latest.latest(path=self.path)
        self.assertEqual(latest_file, expected,)

    def test_latest_with_args(self):
        """
        Ensure that calling the function with arguments of 2 and some
        directory returns the two most recently-created files in the directory.
        """
        expected = set([self.path + "file.new",
            self.path + "file.bak"])
        latest_files = set(latest.latest(2, self.path))
        self.assertEqual(latest_files, expected)

    def tearDown(self):
        for fn in self.file_names:
            os.remove(self.path + fn)

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

Save it. You can't run the tests just yet—you need to have something to test first. The **TestLatest** class, above, defines two tests with common **setUp** and **tearDown**. The **setUp** will take a little longer than our previous tests, because it needs to create three files with different creation times, and it *sleeps* for a second after setting up each file.

Note If you want to use these tests in a different location, change the code to suit the local environment by modifying the PATHSTEM assignment.

Your unit tests show that your function should be able to take in arguments for the number of recent files that you want returned, and the path where it will look for your files. It should also work if you let your function use its default arguments.

Now, let's create the **latest.py** module for the test module to import:

CODE TO TYPE:

```
import glob
import os

def latest(num=1, path="."):
    pass
```

Save it, and then run **test_latest.py**:

OBSERVE: Output from test_latest.py

```
FE
======================================================================
ERROR: test_latest_with_args (__main__.TestLatest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_latest.py", line 34, in test_latest_with_arg
s
    latest_files = set(latest.latest(2, self.path))
TypeError: 'NoneType' object is not iterable

======================================================================
FAIL: test_latest_no_number (__main__.TestLatest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_latest.py", line 25, in test_latest_no_numbe
r
    self.assertEqual(latest_file, expected,)
AssertionError: None != ['v:\\workspace\\file.new']

----------------------------------------------------------------------
Ran 2 tests in 6.031s

FAILED (failures=1, errors=1)
```

What's wrong here? In this case, the issue is with the behavior of the stub function. The stub function is returning **None**, but the **test_latest_with_args()** test expects a list back from **latest.latest()**. We can fix that, but how? Pause, ponder, and reflect on that for a minute before going on to the next part...

Okay, now let's see if you can get your tests to pass! Modify **latest.py** as shown:

CODE TO TYPE:

```
import glob
import os

def latest(num=1, path="."):
    ~~pass~~
    return []
```

Save it and run **test_latest.py**.

OBSERVE:

```
FF
======================================================================
FAIL: test_latest_no_number (__main__.TestLatest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_latest.py", line 25, in test_latest_no_numbe
r
    self.assertEqual(latest_file, expected,)
AssertionError: Lists differ: [] != ['v:\\workspace\\python2_Lesso...

Second list contains 1 additional elements.
First extra element 0:
v:\workspace\FileHandling\src\file.new

- []
+ ['v:\\workspace\\FileHandling\\src\\file.new']

======================================================================
FAIL: test_latest_with_args (__main__.TestLatest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\FileHandling\src\test_latest.py", line 35, in test_latest_with_arg
s
    self.assertEqual(latest_files, expected)
AssertionError: Items in the second set but not the first:
'v:\\workspace\\FileHandling\\src\\file.new'
'v:\\workspace\\FileHandling\\src\\file.bak'

----------------------------------------------------------------------
Ran 2 tests in 6.047s

FAILED (failures=2)
```

Excellent! A little modification to the stub makes sure that your tests fail properly—without errors! The default messages from the failed assertions contain lots of detail to help you figure out why your tests are failing.

Now we need to make our tests pass. Edit **latest.py** as shown:

CODE TO TYPE:

```
import glob
import os

def latest(num=1, path="."):
    files_with_dates = []
    files = glob.glob(os.path.join(path, "*"))
    latest_files = []
    for fn in files:
        files_with_dates.append((os.path.getmtime(fn), os.path.abspath(fn)))
    files_with_dates.sort()
    for file_info in files_with_dates[-num:]:
        latest_files.append(file_info[1])
    latest_files.reverse()
    return latest_files
    ~~return []~~
```

The **setUp()** method (which is run before each test) needs to create three files with the right sequence of creation times. The test's **setUp()** method contains a *sleep* to make sure that the files' creation times differ by at least one second.

Save it and run the test. Both tests should pass:

OBSERVE:

```
..
----------------------------------------------------------------------
Ran 2 tests in 6.115s

OK
```

Nice.

The Value of Tests under Refactoring

Another technique used to produce the most recent files is list comprehension. List comprehensions reduce the amount of code in your program.

Note Shorter code is not always better. Less code could lead to decreased readability. Readability is one of the most important attributes of your code, and should only be sacrificed when performance demands it. It's up to you to decide which way to go.

Let's try using list comprehensions. Modify **latest.py** as shown:

CODE TO TYPE:

```
import glob
import os

def latest(num=1, path="."):
    ~~files_with_dates = []~~
    files = glob.glob(os.path.join(path, "*"))
    dated_files = [(os.path.getmtime(fn), os.path.abspath(fn)) for fn in files]
    dated_files.sort()
    latest_files = [f for (d, f) in dated_files[-num:]]
    latest_files.reverse()
    return latest_files
```

The **latest()** function uses a technique called "decorate-sort-undecorate" to achieve its goal. The file paths need to be sorted by date, so it builds a list of **(date, filename)** tuples, which Python can sort more easily (the date is the "decoration" here, because it isn't required in the result, even though it's necessary for sorting.) By default, the tuples are sorted into ascending order, so the paths of the most recent files will be located at the end.

1	file.bak file.new file.old	Original data: **files = glob.glob(os.path.join(path, "*"))**
2	two seconds ago, file.bak one second ago, file.new three seconds ago, file.old	Original data, *decorated* with the file creation times taken from the filestore interface: **dated_files = [(os.path.getmtime(fn), os.path.abspath(fn)) for fn in files]**
3	three seconds ago, file.old two seconds ago, file.bak one second ago, file.new	Decorated data, now sorted in decorator order: **dated_files.sort()**
4	file.old file.bak file.new	Sorted in decorator order, with decorators removed: **latest_files = [f for (d, f) in dated_files[-num:]]**
5	file.new file.bak file.old	Reversed to give "most recent first" as natural order: **latest_files.reverse()**

So, the algorithm (the set of instructions for completing the task) extracts just the filenames of the most recent files, by using the negative index located in this chunk of code:

```
[for file_info in files_with_dates[-num:]]
```

-num makes it go backwards through values of *num*, then reverses the result, placing the most recent files at the beginning. In other words, ***-num*** takes us backwards from end of the list, by *num* elements (for example, zoo[-5:] would start at the end of zoo and move back five elements, then chop from there to the end of the list). So since the list of files was sorted to get the most recently modified ones last, this clips off the *num* most recent files and then shares them in oldest-to-newest order.

When you run your tests, the one-second delay between file creations causes the run to take over six seconds, but the output should be two successful tests.

Save and run it. With the new **latest** module, your tests still pass. All is well, and you can move ahead feeling confident that nothing is broken (or at least nothing that you're testing for is broken).

Getting a Handle on Files

I'm glad to see you're becoming familiar with some of Python's high-level file handling features: the **glob** module and **os.path**. To reiterate, the **glob** module helps you to search for files using patterns, while **os.path** helps to retrieve file information, used to do various path name acrobatics—like getting the file name out of a full path or splitting and joining path names.

Now, what do pickles and shelves have in common? We'll find out in the next lesson—see you there!

Persistent Storage

Persistent Storage

Python has modules that let you save Python objects. Saving an object actually takes two steps: serialization and persistence. *Serialization* (sometimes called marshaling) is the process of converting an object into a stream of bytes. The stream of bytes can be a textual or binary representation of the original object. *Persistence* means saving that representation to some sort of data store that lives beyond your program's execution time or interactive shell session. Keep in mind that before you *persist* an object, it must be serialized. In this lesson, we'll explore these object serialization and persistence modules:

- pickle
- shelve
- json

Object Serialization and Persistence Using the pickle Module

Python's **pickle** module allows you to serialize objects and save them to a file. When using this module, *pickling* refers to serialization and *unpickling* refers to deserialization. You can pickle the following data types:

- None, True, False
- integers, floating point numbers, complex numbers
- strings, bytes, bytearrays
- tuples, lists, sets, and dictionaries containing only pickleable objects
- built-in functions
- functions defined at the top level of a module (not nested within another class or function)
- classes that are defined at the top level of a module (not nested within another class or function)
- instances of such classes whose __dict__ or __setstate__() is pickleable

Let's try using **pickle**. We'll use pickle's **dump()** function to serialize a number of objects and store them to the disk in the first session. Create a **PersistentStorage** project and assign it to your **Python2_Lessons** working set.

In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> line1 = ["one", 2, 3.0]
>>> line2 = {"dict1": {"random": "stuff"}, "dict2": 2.0}
>>> f = open(r"v:\workspace\PersistentStorage\src\pickle1.pkl", 'wb')
>>> pickle.dump(line1, f)
>>> pickle.dump(line2, f)
>>> pickle.dump(None, f)
>>> f.close()
>>>
```

In the session above, you created a file (written in binary mode, so that the interpreter wouldn't modify the content) and wrote three objects to it with **pickle.dump()**. In each **dump()** statement, the first argument is the object to dump, and the second argument is the file to which the serialized version should be written. Now we'll use the **pickle.load()** function to read the serialized object back from the file. To demonstrate that the file we just created really is permanent, close your current interactive interpreter console (click the red "Terminate" square 🟥) and open a new one for the next part of the exercise. Now, you can be sure that you're seeing exactly what another user would see.

Note You can also see the **pickle1.pkl** file we just created in the Package Explorer window. Select the **PersistentStorage/src** folder and (if necessary) press the **F5** key to refresh the view.

In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> f = open(r"v:\workspace\PersistentStorage\src\pickle1.pkl", 'rb')
>>> for i in range(3):
...     o = pickle.load(f)
...     print(o)
...
['one', 2, 3.0]
{'dict1': {'random': 'stuff'}, 'dict2': 2.0}
None
>>> f.close()
>>>
```

When you open the files, the 'b' option is appended to the mode to deal with the files in binary mode. This is necessary to ensure that a pickle can be moved from one computer to another with a different architecture (say, from an Intel-based machine to a Power PC). In the fileops example from the previous lesson, you serialized data into a text format, but the pickle module in Python 3 uses a binary format by default. You can take a peek at this format by calling **read()** on an open pickle file.

You can also see from our example that it's possible to pickle several items, one after the other, to a file, and then read them by repeated calls of the **pickle.load()** function. If you try to read past the end of the file, **pickle.load()** raises an **EOFError** exception. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> open(r"v:\workspace\PersistentStorage\src\pickle1.pkl", 'rb').read()
b'\x80\x03]q\x00(X\x03\x00\x00\x00oneq\x01K\x02G@\x08\x00\x00\x00\x00\x00\x00e.\x80\x03
}q\x00(X\x05\x00\x00\x00dict1q\x01}q\x02X\x03\x00\x00\x00barq\x03X\x03\x00\x00\x00bazq\
x04sX\x05\x00\x00\x00dict2q\x05G@\x00\x00\x00\x00\x00\x00\x00u.\x80\x03N.'
>>>
```

This binary format is actually pretty compact, especially for more complex data structures. The trade-off is that it's not very human readable. We have omitted some of the text to avoid putting a single, very long line in the listing, which would have made it even more difficult to read. Unlike your fileops module data, which was easy to understand as text, editing our latest file by hand would be highly impractical. Programs in other languages probably won't be able to read this format, because it's been designed exclusively for Python use.

In fact, some older versions of Python might not be able to read this format. There are actually four different pickle protocols—versions 0 through 3. Version 3 is the default protocol used when pickling an object in Python 3, and it's the one that's currently recommended. You can, however, specify which protocol to use as a third argument to the **dump()** function. If you're curious about which formats your version of Python can read, or determining your current default format, that information can be found in the pickle module. More readable information about a pickle file is located in the **pickletools** module. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> import pickletools
>>> pickle.format_version
'3.0'
>>> pickle.compatible_formats
['1.0', '1.1', '1.2', '1.3', '2.0', '3.0']
>>> f = open(r"v:\workspace\PersistentStorage\src\pickle1.pkl", 'rb')
>>> pickletools.dis(f)
    0: \x80 PROTO      3
    2: ]    EMPTY_LIST
    3: q    BINPUT     0
    5: (    MARK

    6: X        BINUNICODE 'one'
   14: q        BINPUT     1
   16: K        BININT1    2
   18: G        BINFLOAT   3.0
   27: e        APPENDS    (MARK at 5)
   28: .    STOP
highest protocol among opcodes = 2
>>> pickletools.dis(f)
   29: \x80 PROTO      3
   31: }    EMPTY_DICT
   32: q    BINPUT     0
   34: (    MARK
   35: X        BINUNICODE 'dict1'
   45: q        BINPUT     1
   47: }        EMPTY_DICT
   48: q        BINPUT     2
   50: X        BINUNICODE 'random'
   58: q        BINPUT     3
   60: X        BINUNICODE 'stuff'
   68: q        BINPUT     4
   70: s        SETITEM
   71: X        BINUNICODE 'dict2'
   81: q        BINPUT     5
   83: G        BINFLOAT   2.0
   92: u        SETITEMS   (MARK at 34)
   93: .    STOP
highest protocol among opcodes = 2
>>> pickletools.dis(f)
   94: \x80 PROTO      3
   96: N    NONE
   97: .    STOP
highest protocol among opcodes = 2
>>>>>> f.close()
```

In our example, pickle has no problem with native data types. The output from **pickletools.dis()** gives us some insight into the way the module stores the data structures, but you don't need to understand serialization format to be able to pickle things. So, what if you wanted to pickle an instance of a class that you wrote? Let's give it a try. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> class Example:
...     def __init__(self):
...         self.item1 = None
...     def item2(self):
...         return "instance variable item1 is %s" % (self.item1)
...
>>> sample1 = Example()
>>> sample1.item1 = "a string"
>>> sample1.item2()
'instance variable item1 is a string'
>>> f = open(r'v:\workspace\PersistentStorage\src\sample1.pkl', 'wb')
>>> pickle.dump(sample1, f)
>>> f.close()
```

So far, your **sample1.pkl** file contains the serialized instance of the **Example** class.

🟥 Now, terminate the console session and open a new interactive one (this is important—you don't want the class definition to continue to be available from your previous session) and try unpickling the **Example** instance. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> f = open(r'v:\workspace\PersistentStorage\src\sample1.pkl', 'rb')
>>> sample1 = pickle.load(f)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Python31\lib\pickle.py", line 1356, in load
  encoding=encoding, errors=errors).load()
AttributeError: 'module' object has no attribute 'Example'
>>>
```

What happened here? You can definitely pickle an object instantiated from your own class, but trying to load your pickled object caused an exception. So, classes that are defined at the top level of a module—that is, classes that are not defined in another class or function—can be pickled.

pickle does not include the actual code of the class used to create the instance when serializing an object, it only includes a reference to the class and the module from where it originated. The original module where the class was defined must be exportable into the unpickling environment.

In the listing above, the class **Example** couldn't be found because it was defined in a previous interactive shell session, so **sample1** was identified as an instance of class **__main__.Example**. The unpickling module was correctly named **"__main__"** (as all interactive sessions are), but there was no class **Example** there.

We'll fix the error by writing the class in a module that can be imported from your interactive shell sessions. To avoid having to tinker with your Python path, create your module and start your interactive shell session in the same path. Everything should work if you create **example.py** in the **PersistentStorage/src** directory. Type the code below as shown:

CODE TO TYPE:

```
class Example:
    def __init__(self):
        self.item1 = None
    def item2(self):
        return "instance variable item1 is %s" % (self.item1)
```

Now you have the **Example** class available in a module. You can use it to create a pickle file in an interactive session. After you've written the pickle file out, you can use **pickletools** as before to see the class encoded in the file. The module and class names appear together. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> from example import Example
>>> obj = Example()
>>> obj.item1 = "some text"
>>> obj.item2()
'instance variable item1 is some text'
>>> obj
<example.Example object at 0x00E51ED0>
>>> import pickle
>>> f = open(r'v:\workspace\PersistentStorage\src\sample1.pkl', 'wb')
>>> pickle.dump(obj, f)
>>> f.close()
>>> f = open(r'v:\workspace\PersistentStorage\src\sample1.pkl', 'rb')
>>> import pickletools
>>> pickletools.dis(f)
    0: \x80 PROTO      3
    2: c    GLOBAL     'example Example'
   19: q    BINPUT     0
   21: )    EMPTY_TUPLE
   22: \x81 NEWOBJ
   23: q    BINPUT     1
   25: }    EMPTY_DICT
   26: q    BINPUT     2
   28: X    BINUNICODE 'item1'
   38: q    BINPUT     3
   40: X    BINUNICODE 'some text'
   54: q    BINPUT     4
   56: s    SETITEM
   57: b    BUILD
   58: .    STOP
highest protocol among opcodes = 2
>>>
```

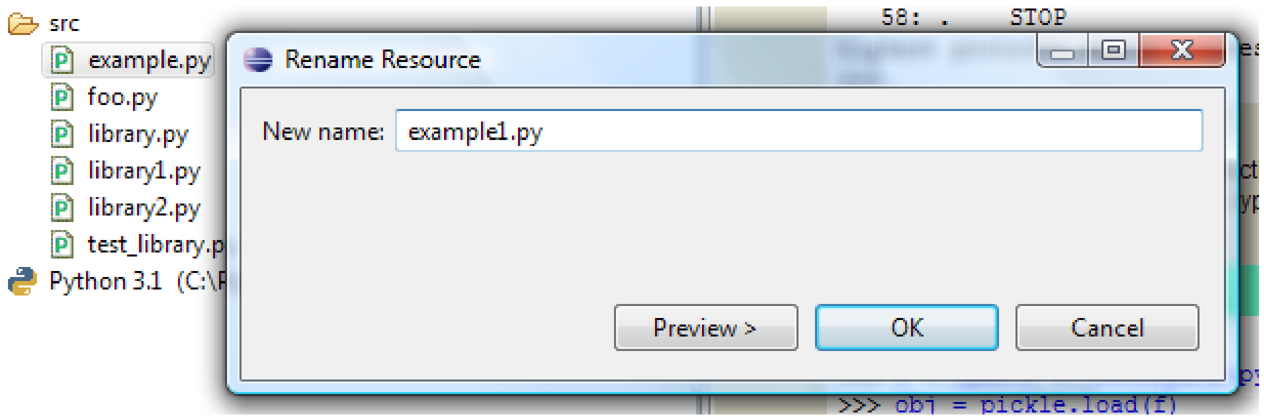
🟥 Again, you'll want to terminate the interactive session and start a new one to make sure that the next session is completely isolated from earlier sessions. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> f = open(r'v:\workspace\PersistentStorage\src\sample1.pkl', 'rb')
>>> obj = pickle.load(f)
>>> f.close()
>>> obj
<example.Example object at 0x00E51CD0>
>>> obj.item1
'some text'
>>> obj.item2()
'instance variable item1 is some text'
>>> import sys
>>> sys.modules['example']
<module 'example' from 'V:\workspace\PersistentStorage\src\example.py'>
>>>
```

You can see from the value of **sys.modules['example']** that the **example** module was imported when the class description was unpickled. The pickle contains the name of the module from which the class was imported, and the interpreter repeats the import to make sure that the required class is available.

Now change the **example.py** file name to **example1.py**, so it will not be importable under the same name. Do this using the context menu—move the cursor over your **example.py** file in the Pydev Package Explorer window, right-click the filename and select **Refactor | Rename**. Enter the new name **example1.py** and click OK.

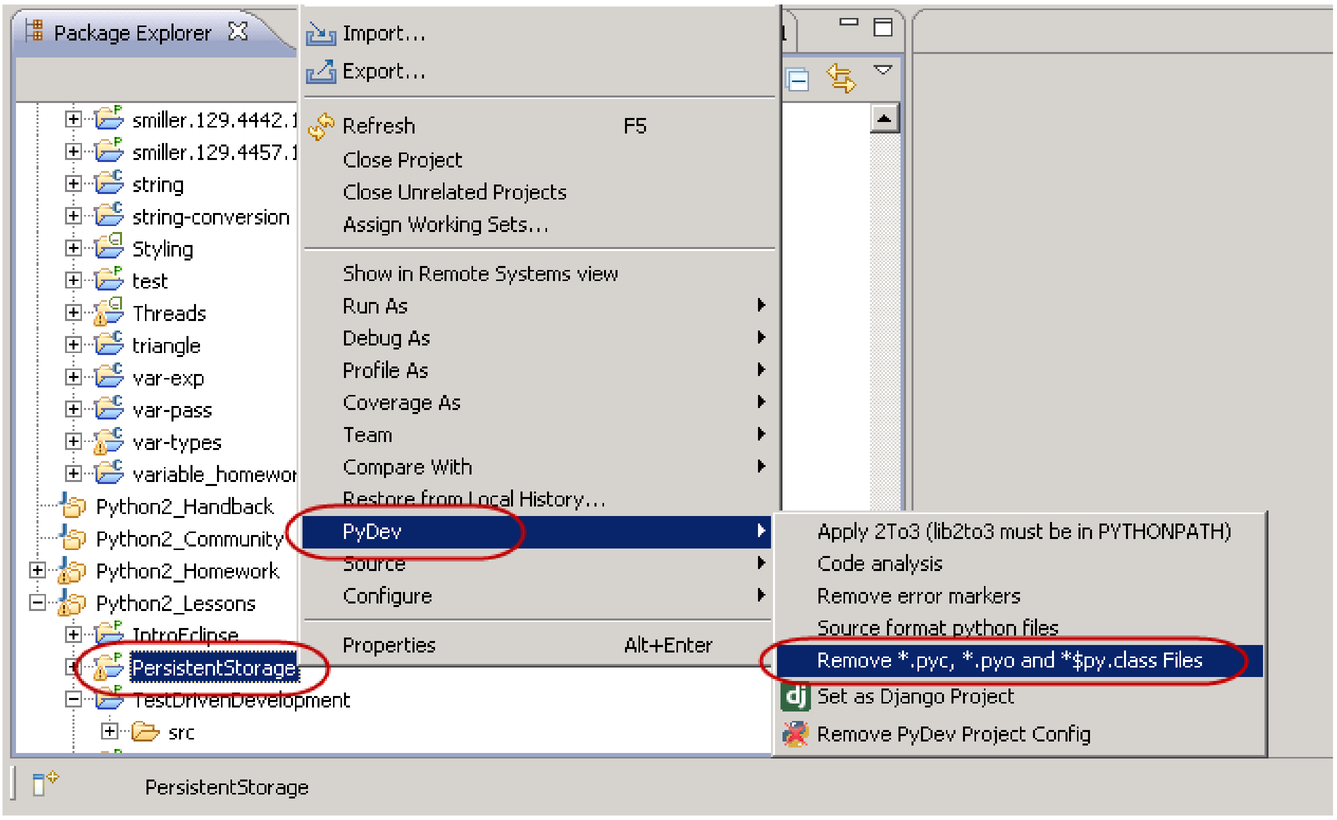


If you repeat the unpickling from the previous session, you will see that it still works, despite renaming the file. Type these commands in an interactive Python console:

INTERACTIVE SESSION:

```
>>> import pickle
>>> f = open(r'v:\workspace\PersistentStorage\src\sample1.pkl', 'rb')
>>> obj = pickle.load(f)
>>>
```

Why does this still succeed? When a module is imported, the interpreter creates a compiled Python file, and even though you have renamed **example.py**, the **example.pyc** file still exists. This is enough for the interpreter to import the **example** module. You have to make sure that the compiled version of the file under the original name is removed. Right-click the **PersistentStorage\src** directory, and select **PyDev | Remove *.pyc, *.pyo and *$py.class files**. You'll have to confirm the actions, after which Ellipse will tell you how many files it has deleted (don't worry if there is more than one - the interpreter can re-create files as necessary).



Finally, start another new Python console and repeat the unpickling from the last session. In the new interactive Python window, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> f = open(r'v:\workspace\PersistentStorage\src\sample1.pkl', 'rb')
>>> obj = pickle.load(f)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Python31\lib\pickle.py", line 1365, in load
    encoding=encoding, errors=errors).load()
ImportError: No module named example
>>>
```

The interpreter can no longer unpickle the object, because it cannot locate the module that defines the required class.

So far, we have used functions from the **pickle** module to handle the pickling and unpickling of objects. The module also defines a **Pickler** class, which lets us create objects. The next example session shows what happens when we try to unpickle too many objects from an **Unpickler** instance. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import pickle
>>> b = ['teeter', 'totter']
>>> a = {'mytoy': b}
>>> f = open(r"v:\workspace\PersistentStorage\src\sample1.pkl", "wb")
>>> pickler = pickle.Pickler(f)
>>> pickler.dump(a)
>>> pickler.dump(b)
>>> f.close()
>>> ff = open(r"v:\workspace\PersistentStorage\src\sample1.pkl", "rb")
>>> unpickler = pickle.Unpickler(ff)
>>> aa = unpickler.load()
>>> bb = unpickler.load()
>>> aa
{'mytoy': ['teeter', 'totter']}
>>> bb
['teeter', 'totter']
>>> aa['mytoy'] is b
False
>>> extra = unpickler.load()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
EOFError
>>>
```

The **Pickler** and **Unpickler** classes are alternatives to calling the **dump()** and **load()** functions directly from the **pickle** module. You can instantiate a Pickler object by passing a file object into the Pickler constructor. From there, you can call the instance's own **dump()** method to store objects into the same file over and over. The **Unpickler** class has a corresponding **load()** method that unpickles objects from the given file sequentially. When we tried to unpickle more objects than were present in the file, the **EOFError** was raised.

The shelve Module

Using Pickler and Unpickler classes allows us to store multiple objects in a single file. Although pickling individual objects with these classes is fairly straightforward, storing and retrieving multiple objects in one file is not completely documented, and the interface is limited (retrieving objects has to be done sequentially, and there's no obvious way to determine how many objects are pickled). An alternative is to use the **shelve** module to create a "shelf," which is a persistent dictionary of objects.

You can store objects in a shelf using a key, and then retrieve them with the same key, just like you would with a dictionary. The keys *must* be encodable as strings—anything else will raise an exception—but the values can be anything that can be pickled (**shelve** uses pickle as its underlying mechanism for serializing objects). Although it has a good interface for storing and retrieving objects, keep in mind that the shelf contents are stored on disk, not in memory, as are *copies* of the objects.

To create a shelf object, pass a file name to the **shelve.open()** function. If the file doesn't exist, it will be created for you as an empty shelf. The shelf object resulting from the call to **shelve.open()** can be used like a dictionary. Use keys to store and retrieve objects. Keys that don't exist will raise an exception. The example below uses the Example class from the example module that you created earlier in this lesson. Make sure you start the interactive shell in the path where that module lives. Before your proceed, rename **example1.py** back to **example.py**. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import shelve
>>> from example import Example
>>> a = [1, 2, 3]
>>> b = Example()
>>> b.item1 = 'some text'
>>> a
[1, 2, 3]
>>> b
<example.Example object at 0x00E677D0>
>>> b.item2()
'instance variable item1 is some text'
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf.shlf')
>>> shelf['a'] = a
>>> shelf['b'] = b
>>> shelf.close()
```

Terminate the console and start a new one. In the new interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import shelve
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf.shlf')
>>> shelf['a']
[1, 2, 3]
>>> shelf['b']
<example.Example object at 0x00EF14B0>
>>> shelf['b'].item2()
'instance variable item1 is some text'
>>> shelf['z']
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\python\lib\shelve.py", line 112, in __getitem__
    f = BytesIO(self.dict[key.encode(self.keyencoding)])
  File "C:\python\lib\dbm\dumb.py", line 124, in __getitem__
    pos, siz = self._index[key]     # may raise KeyError
KeyError: b'z'
>>> shelf.close()
```

If the filename supplied to **open()** does not exist, the file is created. Be careful, though—if a file does exist, you could be writing to a shelf that contains existing objects without knowing it. Also, the filename that you specify is the base filename for the actual file or files that store the shelves' data. Multiple files with various extensions (the ones you usually see are **.dat**, **.dir** and **.bak**) may be created when you use **shelve**, so don't be surprised if you find more files than you initially expected.

Shelve objects do not automatically close themselves; you must explicitly call the **close()** method. However, forgetting to call **close()** does not necessarily mean that your shelve assignments don't get written. Also, indexing into a shelve object yields *a copy of the stored object*, not a reference to the original object. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import shelve
>>> a = [1, 2, 3]
>>> b = ['my', 'random', 'text']
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf2.shlf')
>>> shelf['a'] = a
>>> shelf['b'] = b
>>> shelf.close()
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf2.shlf')
>>> shelf['a']
[1, 2, 3]
>>> shelf['b']
['my', 'random', 'text']
>>> shelf['a'].append(4)
>>> shelf['a']
[1, 2, 3]
>>> a = shelf['a']
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> shelf['a'] = a
>>> shelf['a']
[1, 2, 3, 4]
>>> shelf.close()
>>>
```

One way to update values in a shelf is to take a copy of the object, change the copy, and reassign that new object to the key to persist it. That seems like a lot of code to write for an update!

You can change shelf values more easily by passing an extra keyword argument, **writeback=True**, to shelve's **open()** function. **writeback=True** causes shelve to cache access in memory. When the shelf's **sync()** or **close()** methods are called, the cache is synced back to the actual file. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import shelve
>>> a = [1, 2, 3]
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf3.shlf')
>>> shelf['a'] = a
>>> shelf.close()
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf3.shlf', writeback=
True)
>>> shelf['a']
[1, 2, 3]
>>> shelf['a'].append(4)
>>> shelf['a']
[1, 2, 3, 4]
>>> shelf.sync()
>>> shelf.close()
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf3.shlf')
>>> shelf['a']
[1, 2, 3, 4]
>>>
```

The downside to using **writeback** is that memory usage is high because of the cache used. Also, because all of the writes are performed on either **sync()** or **close()**, those operations will take longer, depending on how many changes need to be written. Finally, as mentioned in the introduction to this section, *shelve does not maintain references when it* *persists objects*. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import shelve
>>> b = ['my', 'random']
>>> a = {'myref':b}
>>> a
{'myref': ['my', 'random']}
>>> b.append('text')
>>> b
['my', 'random', 'text']
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf4.shlf')
>>> shelf['a'] = a
>>> shelf['b'] = b
>>> shelf.close()
>>> shelf = shelve.open(r'v:\workspace\PersistentStorage\src\myshelf4.shlf', writeback=
True)
>>> shelf['a']
{'myref': ['my', 'random', 'text']}
>>> shelf['b']
['my', 'random', 'text']
>>> shelf['b'].append('rules')
>>> shelf['b']
['my', 'random', 'text', 'rules']
>>> shelf['a']
{'myref': ['my', 'random', 'text']}
>>>
```

This makes the shelf a little more like a standard dictionary. That's why many programmers prefer to use shelf in this mode. If your programs terminate in an uncontrolled way, there's a chance that your changes will be lost before they are saved on disk.

Library Project

Now that you've seen some of shelve's capabilities, you can use it to store persistent data in your applications. We'll build a Library class that lets us keep track of books in a persistent data store. We'll also implement methods that let us retrieve a book from our Library class, using its ISBN, title, or author. Let's start with some **tests** to help us look up the books. There is one test method for each of those three ways of retrieving a book.

For the tests to have meaning, there must be a library to hold the test data. Such a library is established in the **setUp()** method, before each test is performed, and then deleted—perhaps a little too enthusiastically—in the **tearDown()** method. Eventually, the library would likely become an external store, but for our test purposes, the "fixture" that the code provides is fine. Create **test_library.py** below as shown:

CODE TO TYPE:

```
import unittest
import library
import os
import glob

class TestLibrary(unittest.TestCase):
    def setUp(self):
        self.lib_fn = r'v:\workspace\PersistentStorage\src\lib.shelve'
        self.lib = library.Library(self.lib_fn)
        self.fixture_author1 = library.Author('Octavia', 'Estelle', 'Butler')
        self.fixture_book1 = library.Book('0807083100', 'Kindred',
            [self.fixture_author1])
        self.fixture_author2 = library.Author('Robert', 'Anson', 'Heinlein')
        self.fixture_book2 = library.Book('0441790348',
            'Stranger in a Strange Land', [self.fixture_author2])
        self.lib.add(self.fixture_book1)
        self.lib.add(self.fixture_book2)

    def testGetByIsbn(self):
        observed = self.lib.get_by_isbn(self.fixture_book1.isbn)
        self.assertEqual(observed, self.fixture_book1)

    def testGetByTitle(self):
        observed = self.lib.get_by_title(self.fixture_book2.title)
        self.assertEqual(observed, self.fixture_book2)

    def testGetByAuthor(self):
        observed = self.lib.get_by_author(self.fixture_book1.authors[0])
        self.assertEqual(observed, self.fixture_book1)

    def tearDown(self):
        self.lib.close()
        shelve_files = glob.glob(self.lib_fn + '*')
        for fn in shelve_files:
            os.remove(fn)

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

In addition to the Library class, there are two other classes in the tests—Book and Author. The Book and Author classes are already implemented. These classes contain some special methods (methods that are surrounded by underscores) that will facilitate the development of your Library class. Implementing the special **__eq__()** method allows objects to be compared using the **==** operator. The **__dict__()** attribute contains all of the attributes of an object. The combination of the __eq__ method and __dict__ can be used to compare two instances of the same class. Implementing **__eq__()** allows you to use the **==** operator to determine whether two instances of an Author or Book object are the same. As you might have guessed, the **!=** operator is handled by the **__ne__()** method.

a == b can be considered equivalent to **a.__eq__(b)**:

```
a == b
   ≡
a.__eq__(b)
```

With Book and Author already written, your job is to implement the library class. Here's a version with stubbed methods. In **library.py**, type the code below as shown:

CODE TO TYPE:

```
import shelve

class Library:
    def __init__(self, fn):
        pass

    def add(self, book):
        pass

    def get_by_isbn(self, isbn):
        pass

    def get_by_title(self, title):
        pass

    def get_by_author(self, author):
        pass

    def close(self):
        pass

class Book:
    def __init__(self, isbn, title, authors):
        self.isbn, self.title, self.authors = isbn, title, authors

    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__
        return False

    def __ne__(self, other):
        return not self.__eq__(other)

class Author:
    def __init__(self, first_name, middle_name, last_name):
        self.first_name, self.middle_name, self.last_name = first_name, middle_name, la
st_name

    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__
        return False

    def __ne__(self, other):
        return not self.__eq__(other)
```

Run your tests; all three should fail:

OBSERVE: Output from test_library.py

```
FFF
======================================================================
FAIL: testGetByAuthor (__main__.TestLibrary)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\PersistentStorage\src\test_library.py", line 29, in testGetByAutho
r
    self.assertEqual(observed, self.fixture_book1)
AssertionError: None != <library.Book object at 0x00B833F0>

======================================================================
FAIL: testGetByIsbn (__main__.TestLibrary)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\PersistentStorage\src\test_library.py", line 21, in testGetByIsbn
    self.assertEqual(observed, self.fixture_book1)
AssertionError: None != <library.Book object at 0x00B83CF0>

======================================================================
FAIL: testGetByTitle (__main__.TestLibrary)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\PersistentStorage\src\test_library.py", line 25, in testGetByTitle
    self.assertEqual(observed, self.fixture_book2)
AssertionError: None != <library.Book object at 0x00BC1550>

----------------------------------------------------------------------
Ran 3 tests in 0.031s

FAILED (failures=3)
```

Use the shelve module to implement the missing features. It's not as much code as you might think. Modify your **Library** class as shown:

CODE TO TYPE:

```
class Library:
    def __init__(self, fn):
        ~~pass~~
        self.fn = fn
        self.shelf = shelve.open(fn)

    def add(self, book):
        ~~pass~~
        self.shelf[book.isbn] = book

    def get_by_isbn(self, isbn):
        ~~pass~~
        return self.shelf[isbn]

    def get_by_title(self, title):
        ~~pass~~
        for book in self.shelf.values():
            if book.title == title:
                return book
        return None

    def get_by_author(self, author):
        ~~pass~~
        for book in self.shelf.values():
            for a in book.authors:
                if a == author:
                    return book
        return None

    def close(self):
        ~~pass~~
        self.shelf.close()

class Book:
    def __init__(self, isbn, title, authors):
        self.isbn, self.title, self.authors = isbn, title, authors

    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__
        return False

    def __ne__(self, other):
        return not self.__eq__(other)

class Author:
    def __init__(self, first_name, middle_name, last_name):
        self.first_name, self.middle_name, self.last_name = first_name, middle_name, la
st_name

    def __eq__(self, other):
        if type(other) is type(self):
            return self.__dict__ == other.__dict__
        return False

    def __ne__(self, other):
        return not self.__eq__(other)
```

All your tests pass. Those passing tests indicate that Book implementation is working. Check it out:

OBSERVE:

```
...
----------------------------------------------------------------------
Ran 3 tests in 2.204s

OK
```

The tests take a significant amount of time to run, whereas before, when all our tests failed, it took almost no time at all. Taking notice of these things during early testing can help you avoid an unpromising line of development (though sometimes you want to proceed anyway, to prove a line of reasoning correct).

The JSON Serialization Format and the json Module

pickle and **shelve** are great for saving objects into persistent storage for other Python programs (that can read and write the same pickle protocol), but there are times when we need to save or transmit objects to programs written in a different language. If we want a human readable, cross-platform and cross-language serialization format, we can use **JSON**. JSON is actually a subset of JavaScript's object literal syntax. Although it was derived from JavaScript, json parsers exist for many languages. In fact, Python 3 comes with a built-in JSON parser.

The full details of the JSON syntax are beyond the scope of this course, but if you take a look at an example, you'll see that it is similar to nested Python lists and dicts. If you want to know more about JSON, visit the JSON website.

OBSERVE: JSON example

```
{
    "foo":"bar",
    "baz":[
        1,
        2,
    ]
}
```

If you wanted to serialize a file object or an instance of your custom class, you would have to define a serialization method or function of your own. Even so, JSON is incredibly useful for exchanging data between programs. You can play around with JSON using Python's **json** module. In an interactive Python console, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import json
>>> a = [1, 2, 3]
>>> b = ['my', 'text']
>>> c = {'a':a, 'b':b, 'none':None, 'true':True}
>>> json.dumps(c)
'{"a": [1, 2, 3], "none": null, "b": ["my", "text"], "true": true}'
>>> d = json.loads(json.dumps(c))
>>> d['a']
[1, 2, 3]
>>> d['b']
['my', 'text']
>>> d['none']
>>> d['true']
True
>>>
```

Just like pickle, the json module has **dump()** and **load()** functions. But you'll notice that in the example, you used **dumps()** and **loads()**—both with an "s" at the end. These methods serialize and unserialize an object to and from the json text format, but rather than persisting an object by writing to a file, or reading from persistent object stores (files), these functions produce and consume strings. Typically, json is used when transmitting or exchanging data over the web. The producers and consumers do not share the same file store; instead they send messages over the network. Consequently, it's more common to serialize objects for transmission, rather than persist them in a file when using the json module.

Note Both **dumps()** and **loads()** functions can be found in the pickle module. They can be used for serialization there, without persistence for content. (The pickle is usually a convenient format when two Python programs communicate).

JSON defines a few primitive data types—strings, numbers, and booleans, as well as objects and arrays. Curly brackets signify an object. Like Python dicts, JSON objects contain a comma-separated list of colon-separated key/value pairs. The values of objects can be any of the types supported by JSON. Arrays, like Python lists, are delimited by square brackets and elements are comma-separated. Like objects, the elements can be of any type supported by JSON. Well-formatted **json** is not difficult to read. But you may already notice a major drawback with this format—it cannot map every Python type. The supported Python-to-JSON data type mappings are:

Python	JSON
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

A Brief Rundown

Serialization means taking a Python object and turning it into a string of bits—either a text or binary format. Deserialization is recreating an object from a text or binary representation of an object. Serialization and deserialization are necessary steps for persistent storage and retrieval of Python objects. Python has a few built-in modules that help deal with serialization and persistence. The pickle module lets you serialize, deserialize, and persist Python objects in a binary format that—for the most part—only Python programs can understand. The shelve module uses the pickle format to store several Python objects using a dictionary-like interface. The json module lets you serialize many of Python's native data types into JSON—a text format that's a subset of JavaScript's object literal syntax. Each serialization and persistence module has its own place. If you're writing a Python application that needs to save a complex data structure's state efficiently (like a game or a text editor), pickle or shelve may be your solution. If you're looking to offer a feed of data to the web, where your clients can be written in any number of various languages, you would use the json module.

Nice job on this lesson! Keep it up. (And you can thank me later for avoiding any of a number of bad pickle joke opportunities.) See you in the next lesson...

Archives

Reading and Writing Archives Using tarfile and zipfile

Python has two modules for handling *archive files*. An archive file is a file that contains an entire directory tree, as well as information about the directory tree itself. An archive file is not a directory; it is a single file which may encapsulate an entire directory tree though, which makes it useful for shipping filestore content from one place to another.

Python supports two archive file formats: zip and tar. Zip files can store compressed versions of files in a directory tree. Tar files are an archival format; they can be compressed using gzip or bzip2. Python can read both regular and compressed tar files (.tar.gz, .tgz, .tar.bz2, or .tbz).

The **zipfile** and **tarfile** modules are used for reading and writing zip and tar files, respectively. Let's take a quick look at these modules; fire up an interactive console. You'll use some of what you learned earlier to prepare a directory to archive. Let's start with **tarfile**. Create the **Archives** project, assign it to the **Python2_Lessons** working set, start an interactive console session, and enter the commands as shown:

INTERACTIVE SESSION:

```
>>> import os
>>> import tarfile
>>> import glob
>>> import shutil
>>> filenames = ["larry", "curly", "moe"]
>>> path = r"v:\workspace\Archives\src\archive_me"
>>> os.mkdir(path)
>>> for fn in filenames:
...     f = open(os.path.join(path, fn), "w")
...     f.close()
...
>>> glob.glob(os.path.join(path, "*"))
['v:\\workspace\\Archives\\src\\archive_me\\larry', 'v:\\workspace\\Archives\\src\\arch
ive_me\\curly', 'v:\\workspace\\Archives\\src\\archive_me\\moe']
>>> archive_fn = r"v:\workspace\Archives\src\archive_me\my_archive.tar"
>>> tf = tarfile.open(archive_fn, "w")
>>> tf.add(path)
>>> tf.close()
>>> tf = tarfile.open(archive_fn)
>>> tf.list()
-rwxrwxrwx root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/
-rw-rw-rw- root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/l
arry
-rw-rw-rw- root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/c
urly
-rw-rw-rw- root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/m
oe
>>> tf.close()
>>> archive_fn_compressed = archive_fn + ".gz"
>>> tf = tarfile.open(archive_fn_compressed, "w:gz")
>>> tf.add(path)
>>> tf.close()
>>> tf = tarfile.open(archive_fn_compressed)
>>> tf.list()
-rwxrwxrwx root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/
-rw-rw-rw- root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/l
arry
-rw-rw-rw- root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/c
urly
-rw-rw-rw- root/root          0 2010-05-28 19:03:27 workspace/Archives/src/archive_me/m
oe
-rw-rw-rw- root/root      10240 2011-05-17 13:50:12 workspace/Archives/src/archive_me/m
y_archive.tar
>>> tf.close()
>>> os.path.getsize(archive_fn)
10240
>>> os.path.getsize(archive_fn_compressed)
209
>>>
```

WARNING

In these examples, we use "*" to add all files in a folder to an archive. If the archive is in the same folder, this can cause a serious problem when you do it again, and repeatedly, because the archive itself will be added to the archive, and you can therefore find youself in an infinite loop creating an infinitely large archive! While it works in our limited examples, you should avoid this practice when you do real work with archives.

Before cleanup, look for these files in the Package Explorer. You may need to refresh the folder view in Package Explorer (right-click the folder name and select **Refresh**).

Then, enter this command in the interactive Python console, as shown:

INTERACTIVE SESSION:

```
>>>shutil.rmtree(path)
>>>
```

Just like the built-in **open()** function, tarfile's **open()** function accepts a file name and a mode. But tarfile's modes are a bit more complicated. In addition to **r**, **w**, and **a** for mode (read, write, and append), you must also consider access type and compression:

Access Type	Symbol	Description
Block Mode	: (colon)	Opens an actual file on disk
Stream Mode	| (pipe)	Opens a stream, socket, or pipe

Compression	Symbol
GZip	gz
BZip2	bz2

Block mode and **no compression** are the defaults. In our example, you used both **w** and **w:gz** to write out your tar files. The second version specifies that your tar file is compressed. At the end of your listing, where you compared the file sizes of the compressed and uncompressed archive file, the compressed version is significantly smaller.

Once you've opened your tar file for writing, you can use its **add()** method to add files to the archive. **add()** can take both filenames and directories, and by default, it adds directories *recursively*—if you have subdirectories in the path that you pass into **add()**, those subdirectories are also added to the archive. You can read tar files by using **open()** in read (**r**) mode. This is the default mode, so in the interactive shell session, we omitted the mode argument. Once you've opened a tar file, you can list its contents with the file's **list()** method. You can also extract its contents using its **extract()** or **extractall()** method.

Also, we used a function called **rmtree()** from the **shutil** module, to remove the directory.

Now we'll take a look at the **zipfile** module. Again, we'll use the file's name and the mode in which we open it to create an interface with zip files. But, instead of a function, the zipfile module offers a **ZipFile** class constructor. In an interactive shell session for zipfile, type the commands below as shown:

INTERACTIVE SESSION:

```
>>> import os, tarfile, glob, shutil, zipfile
>>> filenames = ["groucho", "harpo", "chico"]
>>> path = r"v:\workspace\Archives\src\archive_me"
>>> os.mkdir(path)
>>> for fn in filenames:
...     f = open(os.path.join(path, fn), "w")
...     f.close()
...
>>> glob.glob(os.path.join(path, "*"))
['v:\\workspace\\Archives\\src\\archive_me\\groucho', 'v:\\workspace\\Archives\\src\\ar
chive_me\\harpo', 'v:\\workspace\\Archives\\src\\archive_me\\chico']
>>> archive_fn = r"v:\workspace\Archives\src\archive_me\my_archive.zip"
>>> zf = zipfile.ZipFile(archive_fn, "w")
>>> filenames = glob.glob(os.path.join(path, "*"))
>>> for fn in filenames:
...     zf.write(fn)
...
>>> zf.close()
>>> zf = zipfile.ZipFile(archive_fn)
>>> zf.namelist()
['workspace/Archives/src/archive_me/groucho', 'workspace/Archives/src/archive_me/harpo'
, 'workspace/Archives/src/archive_me/chico', 'workspace/Archives/src/archive_me/my_arch
ive.zip']
>>> #clean up. (Again, you can check the Package Explorer first to see that the files w
ere created.)
...
>>> zf.close()
>>> shutil.rmtree(path)
```

Note This time, for the sake of convenience, we added all of our imports in one line!

One major difference between **tarfile** and **zipfile** is the method used to open the files—with zipfile, we use the class constructor instead of an **open()** method on an instance. As mentioned above, zip archives may contain compressed files. By default, files are stored uncompressed. To compress files, we'd pass a third argument to the class constructor —**zipfile.ZIP_DEFLATED**.

Unlike tarfile's **add()** method, ZipFile's **write()** method does not add files to the archive recursively. That's why we had to use **glob()** to get all of the files before writing them to our archive. (We'd have had to use **os.path.walk** or some similar functionality if there had been subdirectories to process).

You can read in a zip file by passing only the filename to the ZipFile constructor. The **namelist()** method lists all of the files in the archive and, just as in tarfile, the **extract()** method will uncompress and extract the files from the archive. Here's a quick comparison of zipfile and tarfile:

Function	tarfile	zipfile
Open for Writing	tarfile.open(fn, "w")	zipfile.ZipFile(fn, "w")
Open for Writing Compressed	tarfile.open(fn, "w:gz")	zipfile.ZipFile(fn, "w", zipfile.ZIP_DEFLATED)
Open for Reading	tarfile.open(fn)	zipfile.ZipFile(fn)
Add a File to the Archive	tarfile.add(path)	zipfile.ZipFile.write(path)
List Files in an Archive	tarfile.list()	zipfile.ZipFile.namelist()

Extract Files	tarfile.extract()
	or
	tarfile.extractall()
		zipfile.ZipFile.extract()
		or
		zipfile.ZipFile.extractall()

Creating a Recent File Archiver

You can build on **latest.py** to create a function that archives the last modified files in a path. Rather than try to extend the existing **test_latest** module, we'll create another module to test the added functionality. For this test, create a new file named **test_ziplatest.py** in your **Archives** project. The two test modules do have some common features, but for now, we'll write a separate test suite. Enter the code for **test_ziplatest.py** below as shown:

CODE TO TYPE:

```
import unittest
import latest
import time
import os
import shutil
import zipfile

class TestZip(unittest.TestCase):

    def setUp(self):
        self.path = r"v:\workspace\Archives\src\zip_test"
        self.zip_filename = os.path.join(self.path, "test_zip_latest.zip")
        os.mkdir(self.path)
        self.file_names = ["old", "newer", "newest"]
        for fn in self.file_names:
            f = open(os.path.join(self.path, fn), "w")
            f.close()
            time.sleep(1)

    def test_zip_latest(self):
        latest.zip_latest(self.zip_filename, 2, self.path)
        zf = zipfile.ZipFile(self.zip_filename, "w")
        files_in_archive = zf.namelist()
        zf.close()
        observed = set([os.path.basename(f) for f in files_in_archive])
        expected = set(self.file_names[1:3])
        self.assertEqual(observed, expected)

    def tearDown(self):
        os.remove(self.zip_filename)
        try:
            shutil.rmtree(self.path, ignore_errors=True)
        except IOError:
            pass

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

Now, let's make a copy of **latest.py** and stub out a function. To copy the file, go to your **FileHandling/src** project folder, right-click on **latest.py**, and select **Copy**. Then, right-click the **Archives/src** folder and select **Paste**. We'll call the new function **zip_latest()**. Modify the file as shown:

CODE TO TYPE:

```
import glob
import os

def latest(num=1, path="."):
    files = glob.glob(os.path.join(path, "*"))
    dated_files = [(os.path.getmtime(fn), os.path.abspath(fn)) for fn in files]
    dated_files.sort()
    latest_files = [f for (d, f) in dated_files[-num:]]
    latest_files.reverse()
    return latest_files

def zip_latest(fn, num, path):
    pass
```

A quick run will reveal a single failing test:

OBSERVE: Output from test_ziplatest.py

```
F
======================================================================
FAIL: test_zip_latest (__main__.TestZip)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Archives\src\test_ziplatest.py", line 27, in test_zip_latest
    self.assertEqual(observed, expected)
AssertionError: Items in the second set but not the first:
'newest'
'newer'

----------------------------------------------------------------------
Ran 1 test in 3.032s

FAILED (failures=1)
```

Now that the test program has created the zip file, we can change it from write mode to read mode. Edit **test_ziplatest.py** as shown:

CODE TO TYPE:

```
import unittest
import latest
import time
import os
import shutil
import zipfile

class TestZip(unittest.TestCase):

    def setUp(self):
        self.path = r"v:\workspace\Archives\src\zip_test"
        self.zip_filename = os.path.join(self.path, "test_zip_latest.zip")
        os.mkdir(self.path)
        self.file_names = ["old", "newer", "newest"]
        for fn in self.file_names:
            f = open(os.path.join(self.path, fn), "w")
            f.close()
            time.sleep(1)

    def test_zip_latest(self):
        latest.zip_latest(self.zip_filename, 2, self.path)
        zf = zipfile.ZipFile(self.zip_filename, "r")
        files_in_archive = zf.namelist()
        zf.close()
        observed = set([os.path.basename(f) for f in files_in_archive])
        expected = set(self.file_names[1:3])
        self.assertEqual(observed, expected)

    def tearDown(self):
        os.remove(self.zip_filename)
        try:
            shutil.rmtree(self.path, ignore_errors=True)
        except IOError:
            pass

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

Most of the functionality you need is already within your module. Combine what you've learned about archive files with your **latest()**, and add a few lines to **latest.py**, as shown:

CODE TO TYPE:

```
import glob
import os
import zipfile

def latest(num=1, path="."):
    files = glob.glob(os.path.join(path, "*"))
    dated_files = [(os.path.getmtime(fn), os.path.abspath(fn)) for fn in files]
    dated_files.sort()
    latest_files = [f for (d, f) in dated_files[-num:]]
    latest_files.reverse()
    return latest_files

def zip_latest(fn, num, path):
    ~~pass~~
    files_to_archive = latest(num, path)
    zf = zipfile.ZipFile(fn, "w", zipfile.ZIP_DEFLATED)
    for fn_to_archive in files_to_archive:
        zf.write(fn_to_archive)
    zf.close()
```

If the tests pass, your changes to the **latest** module have worked. Congratulations!

Save It in the Archives

Now you've got a good foundation for two archive file formats: zip and tar. We used Python's **zipfile** and **tarfile** modules to read and write each format. Finally, we integrated this knowledge to write a quick function that archived the latest **n** files in a path.

Great work so far! Keep it up!

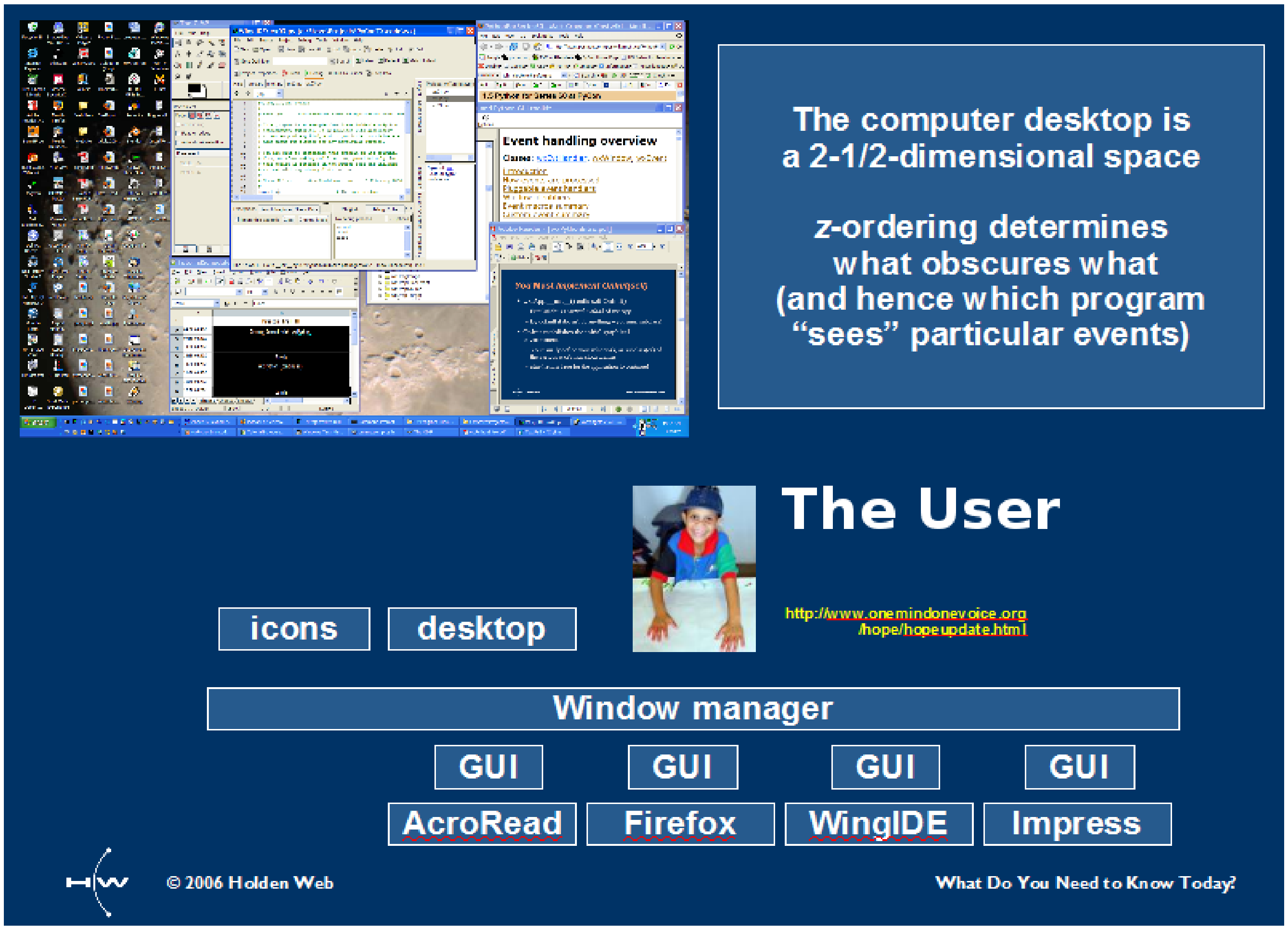
Introduction to Graphical User Interfaces

In this lesson, we'll learn the basics of programming graphical user interfaces (GUIs). GUI-based programs are somewhat different from those you have written so far. You're earlier programs have driven the process of user interaction. When the programs wanted data they prompted the user, and waited for the user to complete their entry by pressing **Enter**.

Consider a program with an interface that has buttons, checkboxes, text entry items, and so on. The user can interact with these elements however they like. But how do we write programs that are ready to respond to whatever the user presents?

The Window Manager

Take a look at the diagram below. The user sees some sort of desktop wallpaper (in this case, an image of the moon) covered with icons, application windows, and (since the desktop is that of a Windows XP machine) the *taskbar* that holds icons representing each running application, a whole load of *quick-launch* icons, and a Start icon that can be used to bring up a menu allowing access to most of the facilities of the computer.



The computer desktop is a 2-1/2-dimensional space

z-ordering determines what obscures what (and hence which program "sees" particular events)

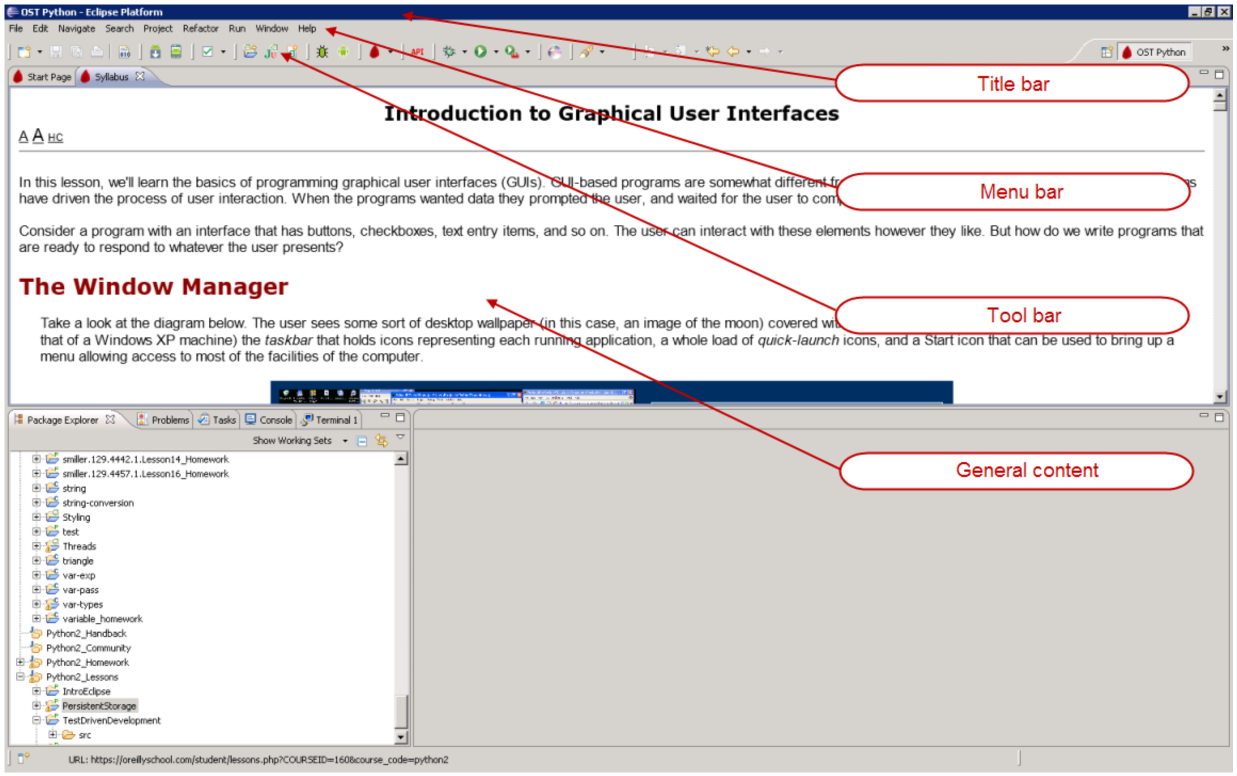
The desktop is called a "two-and-a-half dimensional surface" because, although it does not actually have a third dimension (depth), one window can cover another, just as though it were a piece of paper covering another piece, on a real, physical desktop. (Sadly, my own physical and virtual desktops are rarely tidy!) When you click on something, the *window manager* must know which window is on top where you have clicked, so it can channel the *event* to that window.

In a GUI environment, you write programs that present a description of the desired window structures to the window manager, which is the system component that handles (among other things) tracking mouse movements and distributing keystrokes and mouse clicks to the right programs. Which programs receive these events depends on a number of factors, including the current cursor position and which window has the *focus*.

Each window is composed of widgets, some of which contain other widgets, and so on. One widget can be positioned on top of another. The window manager has to make the determination about which widget is uppermost at the particular position of the cursor when the click occurs. (We'll go over widgets a bit more later in the lesson.)

How Programs Interact with the Window Manager

You are reading this text in the Ellipse teaching environment. There's a *title bar* across the top of the screen. Under the title bar is the Eclipse window's content area, headed by a *menu bar*, under which is the *toolbar*. The toolbar contains a load of buttons, which you can click to make specific things happen.



Introduction to Graphical User Interfaces

In this lesson, we'll learn the basics of programming graphical user interfaces (GUIs). GUI-based programs are somewhat different fr … s have driven the process of user interaction. When the programs wanted data they prompted the user, and waited for the user to com …

Consider a program with an interface that has buttons, checkboxes, text entry items, and so on. The user can interact with these elements however they like. But how do we write programs that are ready to respond to whatever the user presents?

The Window Manager

Take a look at the diagram below. The user sees some sort of desktop wallpaper (in this case, an image of the moon) covered wi … that of a Windows XP machine) the *taskbar* that holds icons representing each running application, a whole load of *quick-launch* icons, and a Start icon that can be used to bring up a menu allowing access to most of the facilities of the computer.

URL: https://oreillyschool.com/student/lessons.php?COURSEID=160&course_code=python2

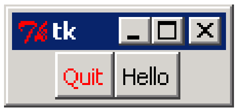
Eclipse needs to know that when you move the cursor over a particular button and click, it has to run the piece of code that corresponds to the function associated with the button. The structure of the window is created by Ellipse when it starts-up and is passed to the window manager, which then triggers specific responses to specific events, calling specific routines. The same is true of any GUI-based program.

All this information is created in a form that the window manager can understand by making calls to a *window library*. The main libraries in Python are *PyQT*, *wxPython*, and *tkinter*. We'll use *tkinter* to explain the principles of working with GUIs. The descriptions of the window structures include references to the specific pieces of code (*event handlers*) that must be run in response to specific events.

The structures can be modified while the program runs. For example, you can arrange for a dialog box to appear when a particular button is clicked. While a program's main window is usually created at the start of the program and continues to exist for the duration of the program, it is not at all uncommon for programs to create and delete other windows as they are required.

Your First Program with a GUI

This example is taken straight from the documentation for the **tkinter** module. The program creates a window that looks like this:



When you click the button on the right, the program prints some text on its standard output. When you click the button on the left, the program terminates. Create a **IntroGUI** project and assign it to your **Python2_Lessons** working set. Then, in the **IntroGUI/src** folder, create a **tkdemo.py** file as shown:

CODE TO TYPE:

```
from tkinter import *

class Application(Frame):
    def say_hi(self):
        print("Hi there, everyone!")

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "Quit"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit
        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
```

Save and run it. Click the **Hello** button and then the **Quit** button, to see what they do.

Let's look at the code more closely:

OBSERVE:

```
from tkinter import *

class Application(Frame):
    def say_hi(self):
        print("Hi there, everyone!")

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "Quit"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit
        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
```

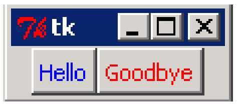
The majority of the code in the program defines a class named **Application**, which subclasses the **tkinter.Frame** class. The **tkinter.Frame** class defines all of the general behaviors required of a program's GUI, but these general behaviors do not encompass the specifics of the contents of this window. For those specifics, we have the **createWidgets()** method.

Let's begin by looking at the **tkinter.Frame** class's **__init__()** method. First, it performs all of the standard **tkinter.Frame** initialization actions by calling its superclass's (**tkinter.Frame's __init__()**) method. Next, it calls the newly created frame's **pack()** method, which prepares it to be part of the window display. Then, it calls the **createWidgets()** method, which as its name suggests, creates the widgets (or components) that go inside of it.

createWidgets() initializes only two widgets: the first is **the Quit button**, which reads **"Quit"** with the foreground (**"fg"**) text in **"red"** and calls the Frame's self.quit() method (inherited from tkinter.Frame) when clicked; the second is the **hi_there** button, which reads "Hello" and calls the Frame's **say_hi()** method when clicked.

Note Hey, wait a minute. In the Python 1 course, didn't we say that we should *never* use the **from *module* import *** form of the import statement? In fact we did. But certain modules have been designed specifically to be used in this way. If **tkinter** were used in the standard form, then our code would be more difficult to read. When writing a typical program, we use many names from **tkinter**. Our code readability is enhanced by limiting the use of qualified names such as **tkinter.Tk**. The **tkinter** module has been designed with that in mind, and although there is always some danger that you might unknowingly overwrite one of the 150+ names it defines, in practice this doesn't happen much.

Now, suppose the customer changed the specification for this project. They want to change the colors and text a bit to make the application to look like this:



The changes include:

- Change the "QUIT" button label to "Goodbye."
- Make the "Hello" label blue.
- Move the "Goodbye" button to the right of the "Hello" button.

Try to make the changes without looking at the answers below.

. . .

Try to figure it out on your own first!

. . .

I mean it!

. . .

Don't peek!

. . .

If everything went alright, your changes look something like those in the box below (additions and changes in this color and deletions in ~~this style~~):

CODE TO TYPE:

```
from tkinter import *

class Application(Frame):
    def say_hi(self):
        print("Hi there, everyone!")

    def createWidgets(self):
        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["fg"]   = "blue"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack({"side": "left"})

        self.QUIT = Button(self)
        self.QUIT["text"] = "Goodbye"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit
        self.QUIT.pack({"side": "left"})

        ~~self.hi_there = Button(self)~~
        ~~self.hi_there["text"] = "Hello",~~
        ~~self.hi_there["command"] = self.say_hi~~
        ~~self.hi_there.pack({"side": "left"})~~

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
```

Creating Widgets in a Window

The **createWidgets()** method creates precisely two widgets, which it stores as the instance attributes **QUIT** and **hi_there**. **Button** is a function defined by the **tkinter** module. When called, it requires the *parent widget* to be provided as the first argument. Since the newly created frame instance (the one whose **__init__()** method is being called) is the parent, **self** is provided as the first argument. This makes the **Application** instance the parent of the button.

Once the **QUIT** widget has been created, the method then sets a number of configuration items. Each of these items has a name and a value:

Item name	Meaning
text	The label to be shown inside the button
fg	The foreground color used to write inside the button (that is, the color in which the text label will be written)
command	The function to call when the button is clicked

The **text** and **fg** configuration items are pretty straightforward. The **command** item takes a little more effort. This particular code is written to allow the creation of multiple windows, each being an instance of the **Application** class. Because the **command** item is an instance method, when the **QUIT** button is clicked on an instance of the Application class, that instance's **quit()** method is called. This method is inherited from the **tkinter.Frame** class, and causes the application to terminate.

Once the widget is fully configured, its **pack()** method is called to place it at the left-hand side of the (containing) application window (other options are "right", "top," and "bottom). That concludes the configuration of the **QUIT** button. Next, a second widget (the **hi_there** button) is created and configured to call the **hi_there** method when it's clicked. This button is then packed to the left of the remaining space in the containing window.

The only other method in the class is **say_hi()**, which is the event handler for clicks on the **hi_there** button. It prints a message on the console whenever it's called by the user.

Top-Level Application Code

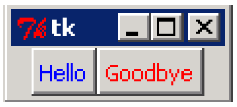
Once the **Application** class is defined, the program needs to create an instance of the application class and pass control to the window manager. The code for that immediately follows the class definition.

The first line, **root = Tk()**, creates the application's main window. If the application created any other windows, they would be children (or grandchildren) of **root**. The next line, **app = Application(master=root)**, creates an instance of the application class (as a subclass of **tkinter.Frame**) and attaches it to the root window.

The call to the application's **mainloop()** method (which is inherited from **tkinter.Frame**) hands control over to the window manager. This method only returns when the application is terminating—the window manager makes direct calls to the event handlers when specific events that have been programmed into the window description occur. Once the application terminates, the program calls its root window's **destroy()** method to release any window manager resources before the program ends.

The Program Window

So, when you run the program, you see a window like this:



The layout of the components was created by calls to the various components' **pack()** methods. Every time you click the "Hello" button, the program will print **"Hi there, everyone!"** in the console window. When you click the "Goodbye" button (or terminate the program by clicking the "X" button at the top right of the window) the program terminates.

So, there you have it. You have written and run your first GUI program using Python's **tkinter** package! Good for you!

Note By the way, you may be wondering what *tkinter* means: *tk* stands for tool kit, and *inter* stands for interface.

Introducing the Tkinter Widget Set

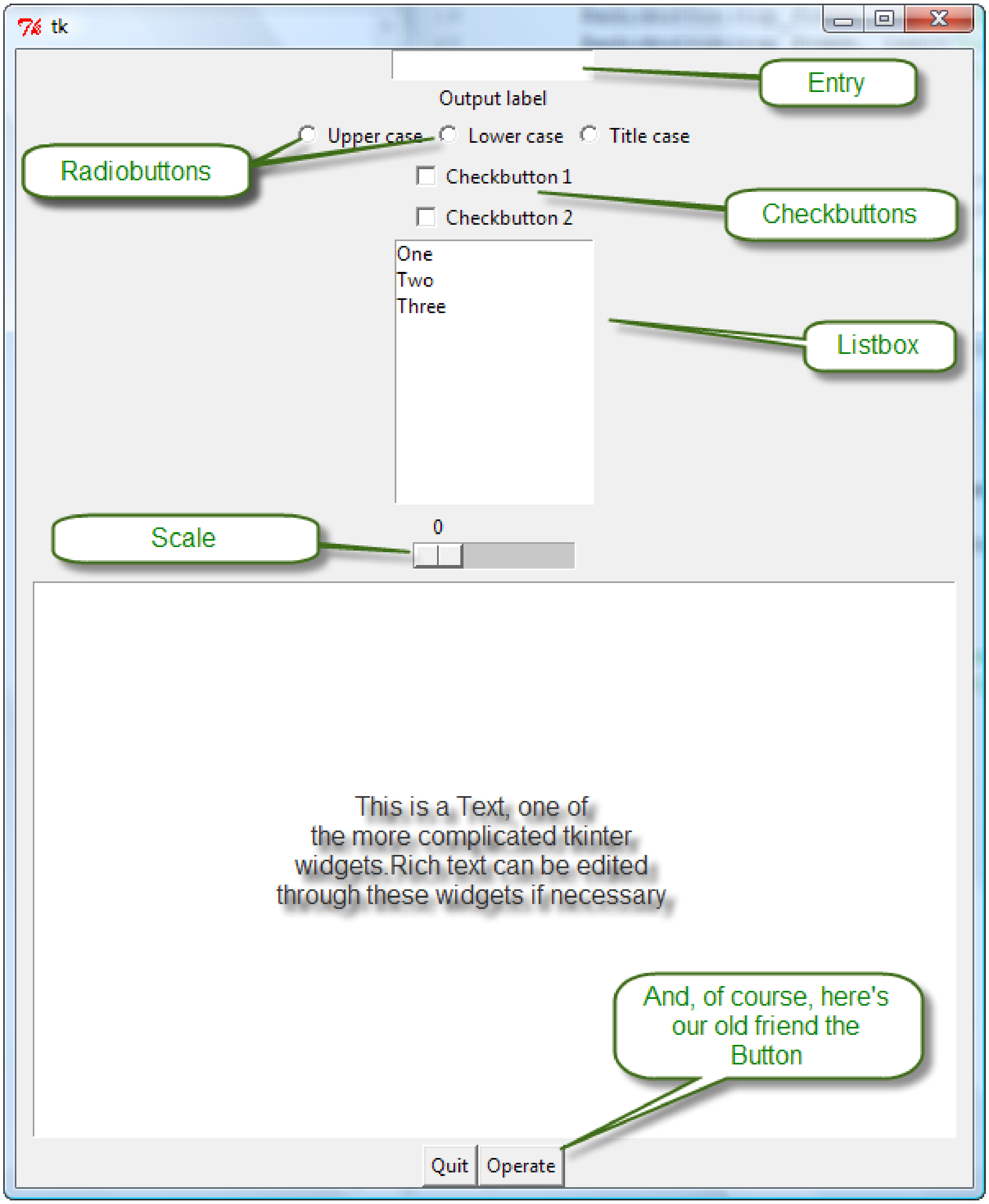
The word "widget" is often used as an abstract name for an object, most often for something manufactured. Modern GUI toolkits, **tkinter** included, are comprised of components that are referred to as "widgets." All **Tkinter** widgets have a lot in common, even though they may not look alike.

There aren't a whole lot of widgets in the **Tkinter** toolkit, but using them wisely will allow you to create a variety of useful graphical interfaces. Below are some important ones that you should know about now:

Widget Type	Purpose
Frame	A container for other widgets. You can set the border and background color, and place other widgets inside of it.
Toplevel	A special kind of Frame that interacts directly with the windows manager. Toplevels will usually have a title bar, and features to interact with the window manager. The windows you see on your screen are mostly top-level windows, and your application can create additional Toplevel windows if it is set to do that.
Button	Users click on buttons to trigger some action. As you already know from the sample program you just entered and ran, clicks on the button can be translated into actions taken by your program (this is actually true of many widgets). Buttons usually have text inside of them, but they can also show graphics.
Checkbutton	A special type of button that has two states; clicking change the state of the button from one to the other.
Label	Labels are used to display pieces of text or images, usually ones that won't change during the execution of the application.

Entry	Used to enter single lines of text and all kinds of input.
Listbox	Used to display a set of choices. The user can select a single item or multiple items from the list. The Listbox can also be rendered as a set of radio buttons or checkboxes.
Scale	Lets the user set numerical values by dragging a slider.
Text	A multi-line formatted text widget, it allows the textual content to be "rich." It may also contain embedded images and Frames.
Message	Similar to a Text, but can automatically wrap text to a particular width, or width and height.
Menu	This is the base widget that you use to put a menu in your window (not all programs need one). It corresponds to the menu bar along the top of your program window, and can also be used to implement "popup" or "context" menus.
Menubutton	Adds choices to your Menus.
Radiobutton	Represents one of a set of mutually exclusive choices. Selecting one Radiobutton from a set, deselects any others.
Scrollbar	Implements scrolling on a larger widget such as a Canvas, Listbox, or Text.
Canvas	A surface on which you can draw graphs and/or plots, and also use as the basis of your own widgets.

Each of the above widgets has its own place in user interfaces. Your first program used a *Toplevel* (created automatically to contain the application) and a *Frame* that contained two *Button*s. In case you are curious about the appearance, here is a picture of a "kitchen sink" interface showing various widgets. By the look of the window, you can probably tell that the elements have been thrown together (in this case, it's the result of a request i received to "show the students what all of these things are"). Try and avoid this look at all cost.


tk
Entry
Output label
Radiobuttons
Upper case
Lower case
Title case
Checkbutton 1
Checkbutton 2
Checkbuttons
One
Two
Three
Listbox
Scale
0
This is a Text, one of the more complicated tkinter widgets.Rich text can be edited through these widgets if necessary
And, of course, here's our old friend the Button
Quit
Operate

Configuring Widgets

So, the program you wrote above runs perfectly well, but the code is a bit to wordy. Each attribute of each widget is configured in a separate statement. If individual aspects of the widgets need to be configured at run-time, this might a convenient way to do it, but when you are creating a widget and many aspects need to be configured, there are better ways.

The most basic way to configure your widgets is with keyword arguments to the widget creation call. Rather than having to write **self.QUIT["fg"] = "red"** after you have created the button, you can add an argument reading **fg="red"** when you create the button. The same principle applies to most other widget configuration items. Try this out by modifying the **tkdemo.py** file as shown:

CODE TO TYPE:

```
from tkinter import *

class Application(Frame):
    def say_hi(self):
        print("Hi there, everyone!")

    def createWidgets(self):
        self.hi_there = Button(self, text="Hello", fg="blue", command=self.say_hi)
        self.hi_there.pack(side="left")

        self.QUIT = Button(self, text="Goodbye", fg="red", command=self.quit)
        self.QUIT.pack(side="left")

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
app.mainloop()
```

Save and run it. The window will have the same appearance and behavior as before, but you've compressed the code considerably without sacrificing readability.

Read over the code; you'll see that once the buttons have been created there's no reference to them anywhere else in the code. So it isn't necessary to save a reference to the buttons in instance attributes, and you could abbreviate the creation of the QUIT button even further to this:

Button(self, text="Goodbye", fg="red", command=self.quit).pack(side="left")

But that might be taking things just a little too far. It's a judgment call. Remember, the programmer who has to understand your code in six months might be you! Ask yourself whether brevity is important enough to make your code that little bit harder to understand.

The config() Method, and Configuration Options

A third way to configure widget options is to call the widget's **config()** method with keyword arguments, naming the options you want to set and giving new values. This is sort of half-way between the two methods you have previously seen, which allows several post-creation changes to be combined into a single statement.

So far, we have used strings as the values of the **pack()** method's **side** parameter. **Tkinter** also provides named constants **LEFT**, **RIGHT**, **BOTTOM**, and **TOP**, which are easier to type and stand out more when you're reading the code. The module provides many similar values that make typing your code easier.

tkinter has many configuration options that you may find confusing at first. Most widgets have a **keys()** method that you can use to learn about the options you can configure. We'll try it out and see how it works. I bet you'll be surprised at how many options are available for configuration. Type the commands below in an interactive session as shown:

INTERACTIVE SESSION:

```
>>> from tkinter import *
>>> b = Button()
>>> for k in b.keys():
...     print(k)
...
activebackground
activeforeground
anchor
background
bd
bg
bitmap
borderwidth
command
compound
cursor
default
disabledforeground
fg
font
foreground
height
highlightbackground
highlightcolor
highlightthickness
image
justify
overrelief
padx
pady
relief
repeatdelay
repeatinterval
state
takefocus
text
textvariable
underline
width
wraplength
>>>
```

There are too many options to consider all of them in detail here (and many that you might never use, even after years of programming with tkinter), but we'll go over the ones you'll use most frequently:

Item name	Definition
background, bg	The color of the body of the widget (on some operating systems, it's impossible to change the background color of some widgets). The colors can be specified as strings (tkinter knows about a lot of colors, and also accepts web-style RGB values like "#006677"—you can read about them here). You can generate these from separate RGB values, where each element is an integer between 0 and 255, using code like: **tk_rgb = "#{0:02X}{1:02X}{2:02X}".format(128, 192, 200)**. If the RGB values are already in a list or tuple, you can use: **tk_rgb = "#{0:02X}{1:02X}{2:02X}".format(*rgb)**
foreground, fg	The color used to write inside the widget, encoded as described above.
padx, pady	The amount of padding to put around the widget, horizontally and vertically. Without this padding, the widget will be just large enough for its contents.

borderwidth	This creates a visible border around a widget.
height, width	Specify the height and the width of a widget (some widgets only let you set the width). Widgets with text in them use a height and width in text units; those containing graphics use a height and width in pixels.
disabledforeground	This specifies the foreground color to use when the widget is *disabled* (that is, when it has been configured not to interact with the user). Most interfaces use gray for disabled foregrounds.
state	The available states depend on the particular widget. The state can be "normal" (as the widget usually looks), "disabled" (how it looks when it won't interact with the user), "active" (how a button looks while the user is interacting with it) or "readonly" (for a Text or Entry widget with text that can be selected, but not changed, by the user). You can use the Tkinter constants NORMAL, DISABLED, and ACTIVE to represent state values as well.

Using More Widgets

Now that you understand a bit more about the way GUIs are put together and the use of widgets, we'll try to use a couple of widgets in an example. We'll create a window that takes a text input and produces different results, depending on which of three radio buttons is selected.

We'll be looking at an interface with inputs—we'll have an Entry widget into which users can type text, and a set of Radiobutton widgets that determine which operation the program performs on the text entered, when the user clicks the **Convert** button.

Reading Widget Values

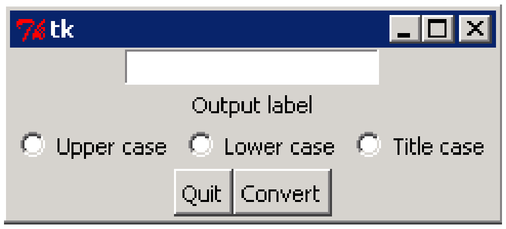
For basic widgets like *Entry* items, you can usually read the item's value by calling its **get()** method, which returns the entered value.

More complex widgets like the Radiobutton can't be handled that way. Radiobuttons come in sets, and only one of them can be selected at a time, so you need to get a value from the set, not from an individual widget. In these cases, we use tkinter Variables; tkinter Variables are associated with widget values. Once the association is made, you can call the Variable's **get()** method instead of the widget's.

Variable types differ according to the type of values you will be extracting. Use a *BooleanVar* for simple yes/no choices, an *IntVar* for integers, a *DoubleVar* for floating-point numbers and a *StringVar* to retrieve text. Those last three are usually associated with an Entry widget, using the special *textvariable* configuration item.

A More Complex Program

At last, here's a program that actually does something!



The next program is longer than previous examples, because it describes a more complicated interface. Two frames are used inside of the main frame. The first contains an *Entry* item where the user can enter text, a *Label* under it, and three *Radiobutton*s. The second frame holds the regular buttons.

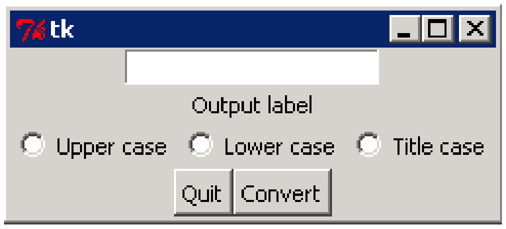
The value of the Entry widget is read from the *text* configuration item, but the Radiobuttons are read using an associated *IntVar*, as integer values are associated with the choices.

Create **texthandler.py** in the **IntroGUI/src** folder and enter the code as shown:

CODE TO TYPE:

```
from tkinter import *

class Application(Frame):
    """Application main window class."""
    def __init__(self, master=None):
        """Main frame initialization (mostly delegated)"""
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        """Add all the widgets to the main frame."""
        top_frame = Frame(self)
        self.text_in = Entry(top_frame)
        self.label = Label(top_frame, text="Output label")
        self.text_in.pack()
        self.label.pack()
        self.r = IntVar()
        Radiobutton(top_frame, text="Upper case", variable=self.r, value=1).pack
(side=LEFT)
        Radiobutton(top_frame, text="Lower case", variable=self.r, value=2).pack
(side=LEFT)
        Radiobutton(top_frame, text="Title case", variable=self.r, value=3).pack
(side=LEFT)
        top_frame.pack(side=TOP)

        bottom_frame = Frame(self)
        bottom_frame.pack(side=TOP)
        self.QUIT = Button(bottom_frame, text="Quit", command=self.quit)
        self.QUIT.pack(side=LEFT)
        self.handleb = Button(bottom_frame, text="Convert", command=self.handle)
        self.handleb.pack(side=LEFT)

    def handle(self):
        """Handle a click of the button by processing any text the
        user has placed in the Entry widget according to the selected
        radio button."""
        text = self.text_in.get()
        operation = self.r.get()
        if operation == 1:
            output = text.upper()
        elif operation == 2:
            output = text.lower()
        elif operation == 3:
            output  = text.title()
        else:
            output = "*******"
        self.label.config(text=output)

root = Tk()
app = Application(master=root)
app.mainloop()
```

Save and run it. You'll see a window that looks like the one shown below. If you click the **Convert** button before you select one of the RadioButtons, then the label text is filled with asterisks. If you make a choice, then the appropriate method is applied to the contents of the Entry widget, and displayed as the text of the label.



So, now you know something about creating GUIs. The code can get pretty lengthy, but it's relatively straightforward. In the next lesson we'll find out more about window layout, which will give us better control over the appearance of our windows.

Further Reading on Tkinter

A lot of the **tkinter** documentation offers code samples written in Python 2. Don't be afraid to get creative in adapting them to Python 3. Python 3 isn't really much different from Python 2 (although the package's name is capitalized in Python 2). I'm confident you'll be able to work out any necessary changes!

Your next port of call should be the Python documentation. The Tkinter Wiki is a community-maintained set of documentation that is informal and friendly to read. It's also user-editable and eternally incomplete; you may want to add your own insights later, as your expertise grows! Onward and forward to the next lesson!

Graphical User Interface Layout

Handling Window Layout

All managers are called as a method call on a widget, with keyword arguments to specify how the widget (which may itself be a container) will be positioned inside of its container.

Managing the way widgets are laid out within their containers (typically frames, although there are other containers) is referred to as "geometry management." The **Tkinter** module has three different ways of packing widgets into their containers. You've already seen the **pack()** method in action. Packing is useful for less complex window layouts, and **pack()** has many options you can use to control how the components are laid out inside their parent frames.

If components are laid out in a regular grid, you can use a widget's **grid()** method instead. If you want to place widgets at specific locations, use the widget's **place()** method. Just make sure you *never* mix calls on **pack()**, **place()**, and **grid()** methods on the same window. This could throw your program into an infinite loop as it tries to satisfy the needs of the more than one different layout scheme.

The Pack Geometry Manager

The table below shows **pack()** method's principal keyword arguments. Most of the values are symbols defined by the tkinter module itself:

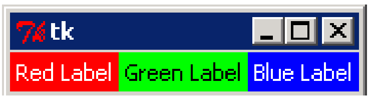
Keyword	Values
fill	X: fill the container in the horizontal dimension. Y: fill the container in the vertical dimension. BOTH: fill the container in both dimensions.
expand	False: the widget is never resized. True: the widget is resized when the container is resized.
side	Specifies which side of the container the widget will be packed against (TOP (the default), LEFT, RIGHT, or BOTTOM).

Let's create a program that demonstrates some of these features. Create a Pydev project named **GUILayout** and assign it to your **Python2_Lessons** working set. In the **GUILayout/src** folder, create a file named **sidebyside.py** as shown:

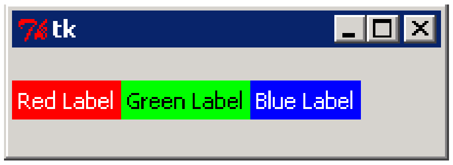
CODE TO TYPE:

```
from tkinter import *

root = Tk()

w = Label(root, text="Red Label", bg="red", fg="white")
w.pack(side=LEFT)
w = Label(root, text="Green Label", bg="green", fg="black")
w.pack(side=LEFT)
w = Label(root, text="Blue Label", bg="blue", fg="white")
w.pack(side=LEFT)

mainloop()
```

When you run the program, you should see a window like this:



Enlarge the window by dragging a corner of it. The labels remain at the left of the window, and are vertically centered in it, like this:

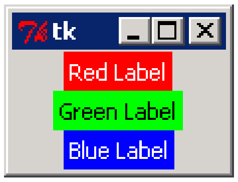


Now, close the window, and change the packing side to TOP as shown:

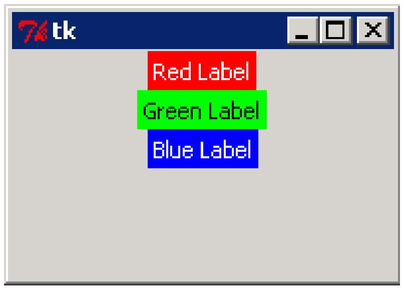
CODE TO TYPE:

```
from tkinter import *

root = Tk()

w = Label(root, text="Red Label", bg="red", fg="white")
w.pack(side=TOP)
w = Label(root, text="Green Label", bg="green", fg="black")
w.pack(side=TOP)
w = Label(root, text="Blue Label", bg="blue", fg="white")
w.pack(side=TOP)

mainloop()
```

Now the program's window shows the labels on top of each other, like this:



Expand the window; the buttons stick to the top and are centered horizontally, like this:

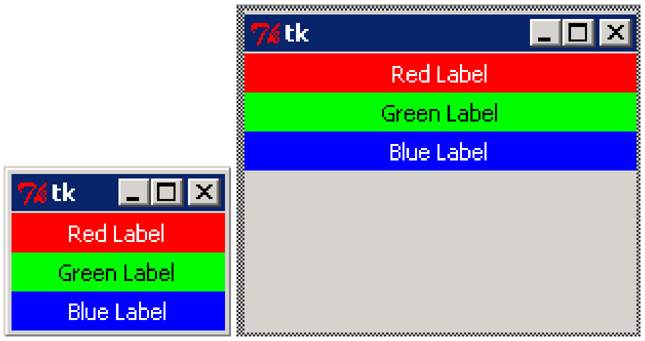


Close the window, and add a **fill=BOTH** argument to each pack call:

CODE TO TYPE:

```
from tkinter import *

root = Tk()

w = Label(root, text="Red Label", bg="red", fg="white")
w.pack(side=TOP, fill=BOTH)
w = Label(root, text="Green Label", bg="green", fg="black")
w.pack(side=TOP, fill=BOTH)
w = Label(root, text="Blue Label", bg="blue", fg="white")
w.pack(side=TOP, fill=BOTH)

mainloop()
```

Now the labels fill the frame. But when you expand the window, the labels only expand horizontally. What's up?

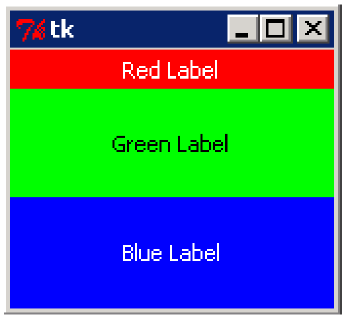


Well, the widgets are not being told to expand, so they only get larger in the dimension where they aren't stacked. So the final change we'll make will be to add an **expand** option to the **pack()** calls (just for fun, we'll omit one to see what happens). Close the window and modify **sidebyside.py** as shown:

CODE TO TYPE:

```
from tkinter import *

root = Tk()

w = Label(root, text="Red Label", bg="red", fg="white")
w.pack(side=TOP, fill=BOTH)
w = Label(root, text="Green Label", bg="green", fg="black")
w.pack(side=TOP, fill=BOTH, expand=True)
w = Label(root, text="Blue Label", bg="blue", fg="white")
w.pack(side=TOP, fill=BOTH, expand=True)

mainloop()
```

When you resize the window, the green and blue labels expand to continue to fill the frame while the red label (which does not have **expand=True**) remains at its original height.



The Grid Geometry Manager

The grid manager is, as its name suggests, most useful when you want components to be laid out on a regular grid. It's probably the most flexible of the managers, and unlike the pack manager, the grid manager does not require you to create a large number of frames to make sure that all of your widgets line up properly as the window is resized.

Once you have created a widget, you can place it in its container in a notional grid, where rows and columns are sized automatically to accommodate the widgets each cell contains, by calling the widget's **grid()** method. An empty row or column will never be displayed or take up any space within the window, which gives you some flexibility about row and column numbering. The table below explains the possible arguments:

Keyword	Values
row	Specifies the row in which this widget should appear.
column	Specifies the column in which this widget should appear.
sticky	Normally a widget appears centered within its cell. The **sticky** attribute can be set to one of four special values, N, S, E, or W, to specify with which side of the cell the widget should be aligned. You can add these values together to cause the widget to expand into its cell. For example, E+W would make expand to occupy the full width of its cell, while N+S+E+W would

	cause the widget to spread out to fill the whole cell.
rowspan, columnspan	If you want a widget to occupy more than one row and/or column, set **rowspan** and/or **columnspan** to the number of rows and/or columns you want it to occupy. The row and column number associated with the widget identify the top-left corner of the spanned block.

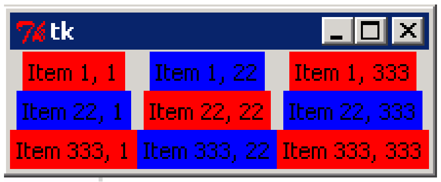
Let's play with the grid manager. In your **GUILayout/src** folder, create a program named **tkgrid.py** as shown:

CODE TO TYPE:

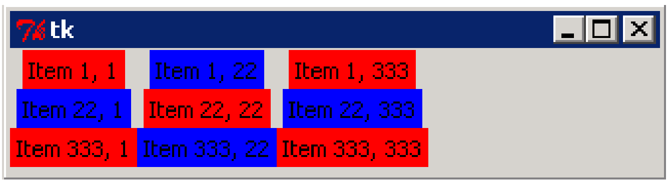
```
from tkinter import *

def colorgen():
    while True:
        yield "red"
        yield "blue"

class Application(Frame):

    def __init__(self, master=None):
        colors = colorgen()
        Frame.__init__(self, master)
        self.grid()
        for r in (1, 22, 333):
            for c in (1, 22, 333):
                txt = "Item {0}, {1}".format(r, c)
                l = Label(self, text=txt, bg=next(colors))
                l.grid(row=r, column=c)

root = Tk()
app = Application(master=root)
app.mainloop()
```

Run the program. It makes the frame rows and columns just big enough for the tallest and widest widgets they contain. Because we chose row and column numbers with different widths, some of the cells have space around them, and you can see the white background of the frame.

Resizing the window demonstrates that only the frame resizes. The cells stay at the top-left corner within the frame.



```
Item 1, 1     Item 1, 22     Item 1, 333
Item 22, 1    Item 22, 22    Item 22, 333
Item 333, 1   Item 333, 22   Item 333, 333
```



```
Item 1, 1     Item 1, 22     Item 1, 333
Item 22, 1    Item 22, 22    Item 22, 333
Item 333, 1   Item 333, 22   Item 333, 333
```

OBSERVE: tkgrid.py

```
from tkinter import *

def colorgen():
    while True:
        yield "red"
        yield "blue"

class Application(Frame):

    def __init__(self, master=None):
        colors = colorgen()
        Frame.__init__(self, master)
        self.grid()
        for r in (1, 22, 333):
            for c in (1, 22, 333):
                txt = "Item {0}, {1}".format(r, c)
                l = Label(self, text=txt, bg=next(colors))
                l.grid(row=r, column=c)

root = Tk()
app = Application(master=root)
app.mainloop()
```

We used an **infinite generator** to create as many alternating colors as the application requires. Calling the **next()** function on a generator is the most convenient way to retrieve the next value in the sequence when you can't iterate over it.

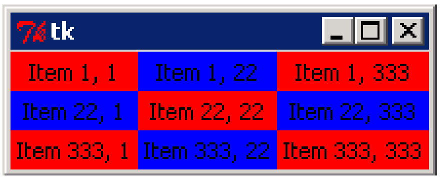
The nested **for** loops create a two-dimensional array where **r** is the row and **c** is the column; the array provides the numbers to display in each grid position AND the display positions themselves (we used multiple-digit numbers to make the text wider for some cells than others; we'd get the same positioning with (1,2,3)).

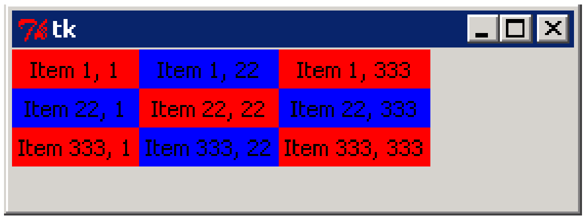
Close the window. The white space issue can be addressed by making the cells sticky on the East and West sides:

CODE TO TYPE:

```
from tkinter import *

def colorgen():
    while True:
        yield "red"
        yield "blue"

class Application(Frame):

    def __init__(self, master=None):
        colors = colorgen()
        Frame.__init__(self, master)
        self.grid()
        for r in (1, 22, 333):
            for c in (1, 22, 333):
                txt = "Item {0}, {1}".format(r, c)
                l = Label(self, text=txt, bg=next(colors))
                l.grid(row=r, column=c, sticky=E+W)

root = Tk()
app = Application(master=root)
app.mainloop()
```

Save and run it. This fixes the white space problem by making all cells in each column the same width. When the window is expanded, however, the rows and columns remain at the top-left of the frame and unchanged in size.





In order to have the columns and rows expand to fill the frame, we actually need to reconfigure the frame itself. A frame with widgets that are configured using the grid manager has **rowconfigure()** and **columnconfigure()** methods, which you can call to apply specific configurations. The first argument is always the row or column index; this can be followed by a number of keyword arguments:

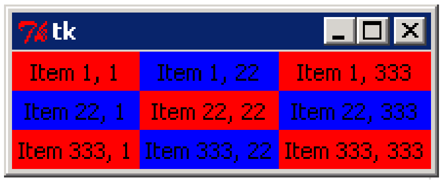
Keyword	Meaning
minsize	Defines the row's or column's minimum size. (Note that the row or column still will not be displayed if there are no widgets present within it.)
pad	Sets the size of the row or column by adding the specified amount of padding to the height of the row or the width of the column.
weight	Determines how additional space is distributed between the rows and columns as the frame expands. The higher the weight, the more of the additional space is distributed between the rows or columns. A row with weight 2 will expand twice as fast as a row with weight 1; it works the same way for columns.

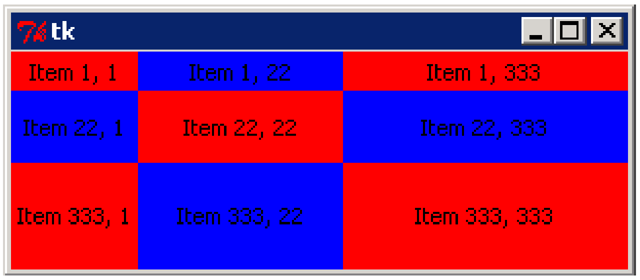
So by calling **rowconfigure()** and **columnconfigure()** methods on the frame, we can fix the second problem. Close the window and modify **tkgrid.py** as shown:

CODE TO TYPE:

```
from tkinter import *

def colorgen():
    while True:
        yield "red"
        yield "blue"

class Application(Frame):

    def __init__(self, master=None):
        colors = colorgen()
        Frame.__init__(self, master)
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0, weight=1)
        self.grid(sticky=W+E+N+S)
        rcount = 0
        for r in (1, 22, 333):
            self.rowconfigure(r, weight=rcount)
            rcount += 1
            ccount = 0
            for c in (1, 22, 333):
                self.columnconfigure(c, weight=ccount)
                ccount += 1
                txt = "Item {0}, {1}".format(r, c)
                l = Label(self, text=txt, bg=next(colors))
                l.grid(row=r, column=c, sticky=W+E+N+S)

root = Tk()
app = Application(master=root)
app.mainloop()
```

Save and run it. The master frame is configured to expand as the program window (a grid of one row and one column) expands. Each row and column is given a weight one higher than the preceding one, starting with zero. This means that as the window expands, the top left cell always stays the same size, and the third row and column expand more than the second.





Close the window.

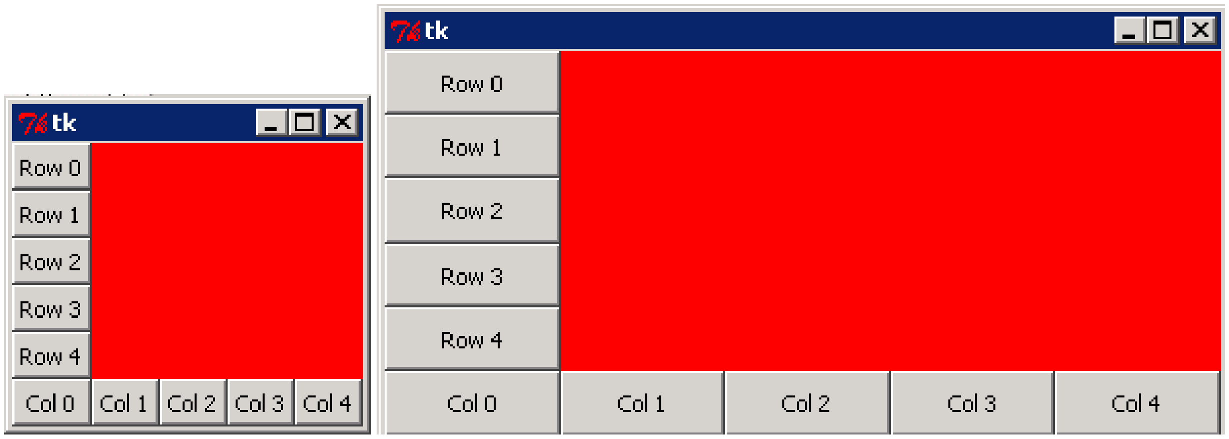
Finally, we're going to see how the **rowspan** and **columnspan** keyword arguments allow us to build flexible layouts. In this case, we'll have a column of buttons on the left, a row of buttons along the bottom, and a frame occupying the remainder of the window. Create **grdspan.py** as shown:

CODE TO TYPE:

```
from tkinter import *

ALL = N+S+W+E

class Application(Frame):

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0, weight=1)
        self.grid(sticky=ALL)
        for r in range(5):
            self.rowconfigure(r, weight=1)
            Button(self, text="Row {0}".format(r)).grid(row=r, column=0, sticky=
ALL)
        self.rowconfigure(5, weight=1)
        for c in range(5):
            self.columnconfigure(c, weight=1)
            Button(self, text="Col {0}".format(c)).grid(row=5, column=c, sticky=
ALL)
        f = Frame(self, bg="red")
        f.grid(row=0, column=1, rowspan=5, columnspan=4, sticky=ALL)
root = Tk()
app = Application(master=root)
app.mainloop()
```

This application again starts out by configuring the frame as a single-row, single-column, expanding grid. Then it configures five buttons in column zero, and adds a sixth row (numbered 5—remember the numbering starts from zero here) containing five buttons. The window has six rows and five columns.

The remainder of the window is occupied by a red Frame; its top-left corner is next to the top button. So it has to span five rows and four columns. When you run your program, the frame should occupy the whole window, even after the program window is resized. Because the buttons are sticky on all four edges, they expand to fill the space the grid manager allocates to them.



The Place Geometry Manager—Don't Use It

We mention this manager only because you might encounter code that uses it. Frankly, the available documentation is insufficient to explain how it works, but you can place a widget either "relatively" (by specifying a **relx** and **rely** argument between 0 and 1 that says how far along the container's width and height the widget should be placed) or "absolutely", specifying an **x** and a **y** position in absolute screen coordinates.

While the place manager allows most flexibility, it is also the most difficult to use, and is outside the scope of this course.

So now you can achieve a required window layout, using either the pack or the grid geometry managers. Excellent! In the next lesson, we'll focus on event handling, and introduce you to a number of tkinter's built-in dialogs. See you there...

More About Graphical User Interfaces

GUI Events

Your program can process several different types of events. The most significant events for most programs are mouse clicks (particularly on buttons) and keystrokes. Some events are processed automatically by the widgets themselves—for example, when you click a radio button or a checkbox, its state is changed automatically without the programmer having to program any specific action. Other events include such things as mouse wheel movements, timers expiring, windows being covered up and exposed, and so on. When you're starting to program GUIs, you can ignore all but the most common events, and let the window manager handle the rest for you.

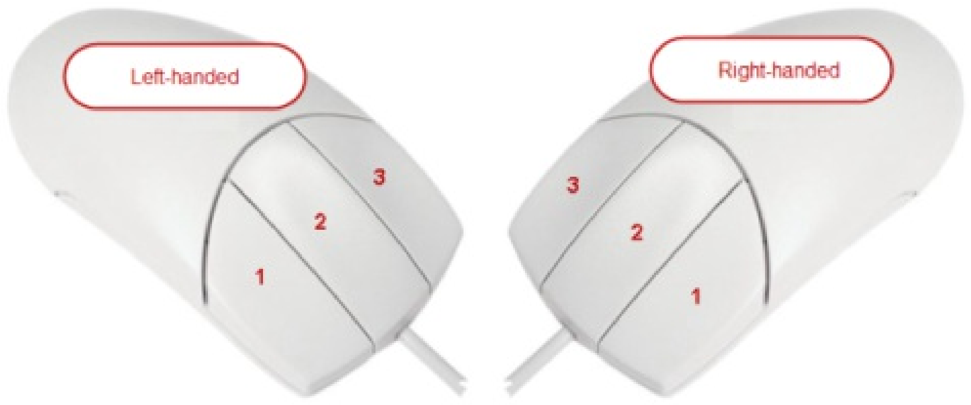
In this lesson, we'll learn how to write programs that respond to events in various ways. You already know how to read and set the values of some widgets. Now you're going to expand your knowledge and learn to create windows on the fly for common tasks like opening and saving files.

Binding Events in tkinter

So far, we've bound event handlers to events using the **command** configuration option with buttons. Many widgets have a **bind()** method that dynamically connects a specific event type to a piece of code in a program. Sometimes you'll need to do this, because not every widget has a natural event to associate with a **command** configuration option.

A widget's **bind()** method has two arguments. The first is the name of the event to be bound, and the second is the handler function to run when the event is detected within the widget. Most events are named using strings starting with "<" and ending with ">." For example, a click of the left mouse button is named "<Button-1>."

For left-handed mice the buttons are reversed, so the same button numbers apply for left-handed users:

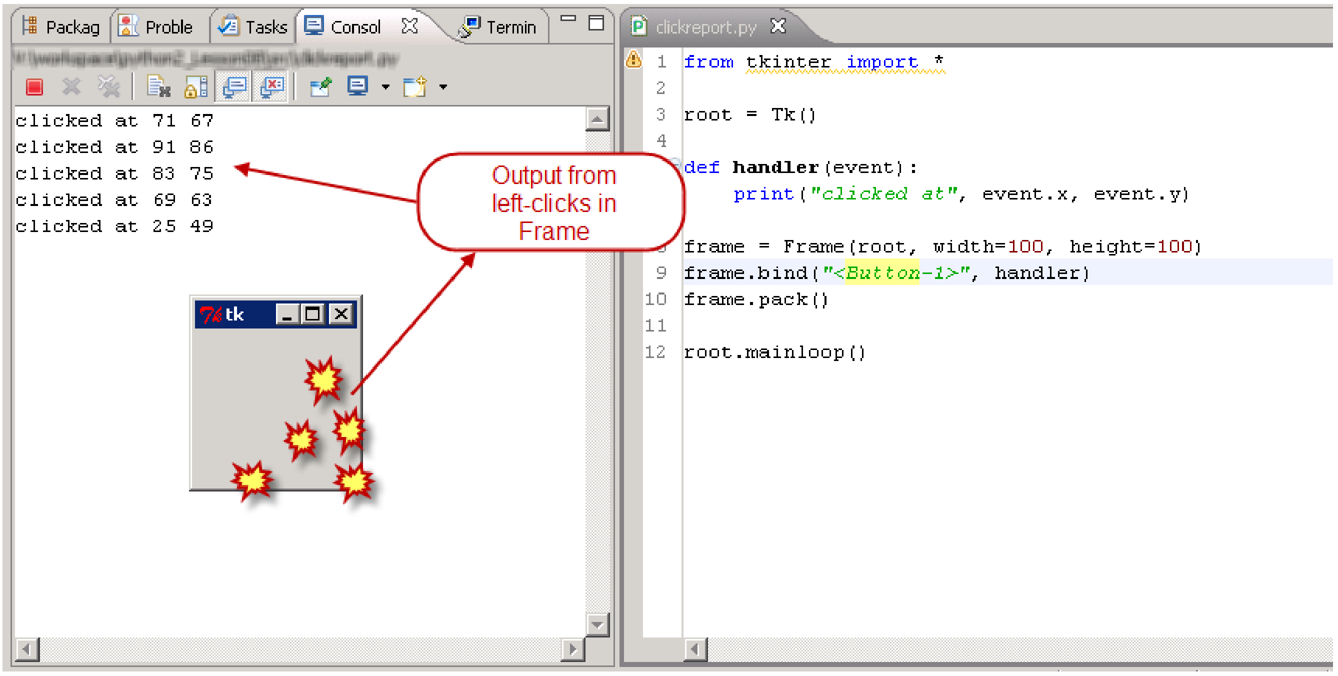


Let's see that in action. Create a new Pydev project named **MoreGUI** and assign it to the **Python2_Lessons** working set. In the **MoreGUI/src** folder, create a program named **clickreport.py** as shown:

CODE TO TYPE:

```
from tkinter import *

root = Tk()

def handler(event):
    print("clicked at", event.x, event.y)

frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", handler)
frame.pack()

root.mainloop()
```

Save and run it. You see something like this:



```
clicked at 71 67
clicked at 91 86
clicked at 83 75
clicked at 69 63
clicked at 25 49
```

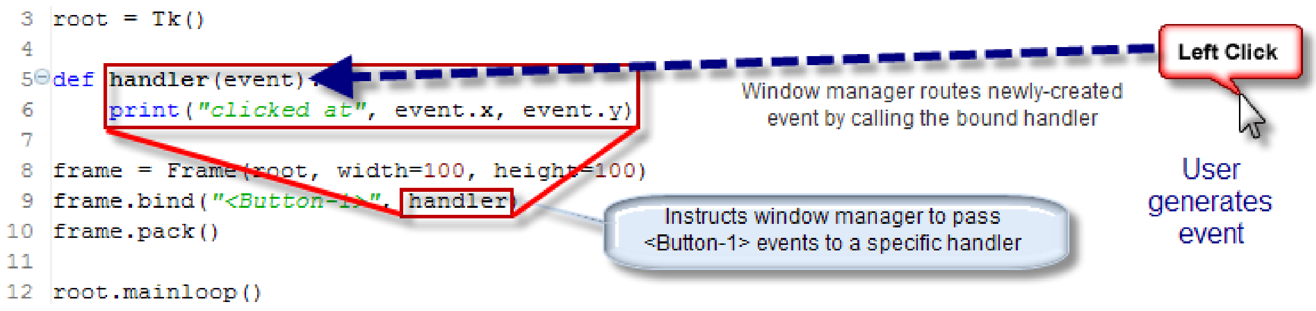
```
from tkinter import *

root = Tk()

def handler(event):
    print("clicked at", event.x, event.y)

frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", handler)
frame.pack()

root.mainloop()
```

Sorry, you won't see those fun little yellow explosions, but each time you left-click the mouse button inside the frame, you'll see a report of the cursor position in the console window.

OBSERVE:

```
from tkinter import *

root = Tk()

def handler(event):
    print("clicked at", event.x, event.y)

frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", handler)
frame.pack()

root.mainloop()
```

Our **handler** is called whenever **<Button-1>** (the left button on a right-handed mouse, or the right button on a left-handed mouse) is clicked.



```
root = Tk()

def handler(event):
    print("clicked at", event.x, event.y)

frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", handler)
frame.pack()

root.mainloop()
```

Left Click
Window manager routes newly-created event by calling the bound handler
User generates event
Instructs window manager to pass <Button-1> events to a specific handler

Notice that, unlike the widget **command** functions, a function bound using a widget's **bind()** method is called with an argument. This argument is an *event* object, and contains information about the specific event that triggered the call to the event handler. In this case, the program extracts the (frame-relative) coordinates of the **<Button-1>** mouse click event and prints those.

Event Objects

The Event object contains data about an event that has just occurred, and it is passed as a single argument to the event handler function. It has several useful attributes (some others are not listed below because they are difficult to use portably):

Attribute Name	Purpose
widget	The widget in which the triggering event occurred. This allows the same function to handle events from multiple widgets.
x, y	The cursor position where mouse events occurred, relative to the top-left corner of the widget in which the event occurred.
x_root, y_root	The cursor position where mouse events occurred, relative to the top-left corner of the screen.
height, width	The new size of the widget (only set for "<Configure>" events).
char	The character code associated with a "<Key>" event.

Mouse Event Names

You'll need to be able to describe events when you ask **tkinter** to establish event bindings. As you saw in the code example above, you can capture a left-click of the mouse with the event name "**<Button-1>**".

You may also run in to code that uses " **<ButtonPress-1>**" or "**<1>**" as a name for the same event. These are equivalent, but we prefer the first form because it's less ambiguous. As you might expect, you can use "**<Button-2>**" and "**<Button-3>**" (and their equivalents) to refer to clicks of the middle and right buttons respectively. You can also detect double- and triple-clicks with **"<Double-Button-*n*>"** and **"<TripleButton-*n*>"** (where *n* is 1, 2, or 3).

You can capture "drag" events—movements of the pointer while a mouse button is held down—with **<B1-Motion>**, and "drop" events with **"<ButtonRelease-1>"** (this applies to buttons 2 and 3 as well).

The **"<Enter>"** event is raised when the pointer enters the screen area occupied by a particular widget, and the **"<Leave>"** event occurs when the pointer leaves the area.

Keyboard Event Names

You can capture the events that occur when the user presses particular keys, using the event name **"<Key>"**. When such an event occurs, the event's **char** attribute tells you which key was pressed (unless it was a special key, like one of the arrow keys or a Shift key. Each of these keys has a special name, which can be used to bind event handlers.

The special key event names are **"<Cancel>"** (the Break key), **"<BackSpace>," "<Tab>," "<Return>,"**(the Enter key) **"<Shift_L>"** (any Shift key), **"<Control_L>"** (any Control key), **"<Alt_L>"** (any Alt key), **"<Pause>," "<Caps_Lock>," "<Escape>," "<Prior>"** (Page Up), **"<Next>"** (Page Down), **"<End>," "<Home>," "<Left>," "<Up>," "<Right>," "<Down>," "<Print>," "<Insert>," "<Delete>," "<F1>"** through **"<F12>," "<Num_Lock>"s**, and **"<Scroll_Lock>."**

Each individual regular key can also be identified by the string containing the character it produces, *without* the surrounding angle brackets. So, to capture a press of the "A" key, use the event name **"A"**. Remember that **"1"** is the name of the event that occurs when the number one (1) key is pressed. But **"<1>"** is a mouse button binding event. If your program concerns just a couple of keystrokes, it's usually easier to bind the individual keystrokes than to bind **"<Key>"** and then analyze each keystroke.

Keyboard Focus

In a windowed user interface, you can change which widget receives keyboard input. The most straightforward way to assign focus to a widget is to click on it, although that also triggers a mouse event. These events are usually ignored by default, although buttons "expect" to be clicked on, and if a button has an associated command function, clicking on the button will cause that function to be called. Different types of widget handle keyboard input in different ways.

Entry widgets accept most characters and insert them into the value returned by the widget's **get()** method. A Radiobutton will only action a space, which is equivalent to selecting that widget from its associated set (automatically clearing any others in the same set). You can also change the focus by pressing the **Tab** key (or **Shifted+Tab** to move in the opposite direction).

Dialog boxes are special cases, with specific behaviors. The **Enter** key is equivalent to clicking the default button in the dialog (which is configured with **default=ACTIVE**) and the **Esc** key is equivalent to clicking the **Cancel** button.

Keyboard Event Handling

When an event is fired by the window manager (for example, when you press a key or click a mouse button) then the event fires first in the component that is "topmost" in the window layout. So when you click a button, since the button is (usually) inside a frame, the click is sent first to the button.

Now, buttons and the other "canned" widgets are special cases, because they make sure that events upon which they take action are never seen by anything "below" them. In general though, this is not so the case. Events are normally distributed to every widget that is part of the hierarchy. So, when you click the mouse on a frame with a parent that is the root window, the click event is delivered first to the frame and then to the root window. It works the same way with keyboard events.

In our next program we'll investigate this feature. Create a new Python file in the **MoreGUI/src** folder, named **evtreport.py** as shown:

CODE TO TYPE:

```
from tkinter import *

root = Tk()

def handler(event):
    print("Keystroke '{0}' ({1}) {2} ".format(event.char, len(event.char), event
.keycode))

frame = Frame(root, width=100, height=100)
frame.bind("<Key>", handler)
frame.pack()
frame.focus()

root.mainloop()
```

Save and run it. Click inside the window, and then try pressing a variety of keys. Most keystrokes are reported. If you look carefully, you'll see that not all keystrokes have a character associated with them. (Which ones don't? It's a challenge to handle these keys in a platform-independent way, because they vary according to hardware and operating systems). If you hold a key down, the automatic repetition associated with doing this are reported as separate keystrokes (even though no physical key movement on the keyboard). If your "Caps Lock" key is like mine, it also repeats despite the lack of physical keystrokes.

OBSERVE: evtreport.py

```
from tkinter import *

root = Tk()

def handler(event):
    print("Keystroke '{0}' ({1}) {2} ".format(event.char, len(event.char), event
.keycode))

frame = Frame(root, width=100, height=100)
frame.bind("<Key>", handler)
frame.pack()
frame.focus()

root.mainloop()
```

frame = Frame(root, width=100, height=100) creates a 100 x 100-pixel frame inside the root window. The **bind()** function captures all keystroke events in the frame (**frame.focus()** ensures that whatever the user types is captured in the frame), and passes them to the **handler**, which prints the keystroke received.

You can associate events with the root window of your application if you like. This is a good way to make sure that an event is trapped no matter which widget it is first presented to (so long as that widget doesn't stop the event from propagating through the widget hierarchy).

If you want to trap only certain keys, you can adjust the program so that other key presses aren't handled. This next modification will do that, handling only lower-case "o" and "k." Modify **evtreport.py** as shown:

CODE TO TYPE:

```
from tkinter import *

root = Tk()

def handler(event):
    print("Keystroke '{0}' ({1}) {2} ".format(event.char, len(event.char), event
.keycode))

frame = Frame(root, width=100, height=100)
~~frame.bind("<Key>", handler)~~
frame.bind("o", handler)
frame.bind("k", handler)
frame.pack()
frame.focus()

root.mainloop()
```

Now, most keystrokes don't result in any reporting whatsoever from your program. Since you bound only specific keyboard events to your frame, the handler is triggered only when those events occur.

Event Propagation

So, what happened to the keystrokes that weren't passed to the handler? Were they not passed to the program, or were they passed to the program and then ignored? Events actually propagate back through a widget to its container, and then to that container's container, and so on, until they reach the root window, unless something specifically stops them from propagating. Many of the standard widgets driven by mouse clicks do stop the clicks from propagating; it would be pretty confusing if a button click had multiple effects! You can allow mouse events to propagate from the widgets you create in much the same way keyboard events are currently propagating to the root window of the frame.

You can see what the root window is receiving by binding events to your application's root window (which is located between the Frame and the window manager) by adding an event binding with a separate handler. Modify **evtreport.py** as shown:

CODE TO TYPE:

```
from tkinter import *

root = Tk()

def handler(event):
    print("Keystroke '{0}' ({1}) {2} ".format(event.char, len(event.char), event
.keycode))

def handler2(event):
    print("RootKeystroke '{0}' ({1}) {2} ".format(event.char, len(event.char), e
vent.keycode))

frame = Frame(root, width=100, height=100)
frame.bind("o", handler)
frame.bind("k", handler)
root.bind("<Key>", handler2)
frame.pack()
frame.focus()

root.mainloop()
```

Now when you type an "o" or a "k," you see *two* events being reported (actually, it's the same event being reported twice). The first report comes from the Frame, and the second from the root window. Other keys are reported only by the root window because they aren't bound in the frame, so the window manager doesn't notify it about those events.

Is there some way to inhibit this propagation of events up through the container hierarchy? In fact, there is. If your handler returns a specific value, the string "break," this tells the event processing portion of the window manager to stop propagating the event. This doesn't just apply to keystrokes, as our final modification to the event reporter program will demonstrate. Modify **evtreport.py** as shown:

CODE TO TYPE:

```
from tkinter import *

root = Tk()

def handler(event):
    print("Keystroke '{0}' ({1}) {2} ".format(event.char, len(event.char), event
.keycode))
    return "break"

def handler2(event):
    print("RootKeystroke '{0}' ({1}) {2} ".format(event.char, len(event.char), e
vent.keycode))

def handler3(event):
    print("Frame clicked at", event.x, event.y)
    if event.x > 50 and event.y > 50:
        return "break"

def handler4(event):
    print("Root clicked at", event.x, event.y)

frame = Frame(root, width=100, height=100)
frame.bind("o", handler)
frame.bind("k", handler)
frame.bind("<Button-1>", handler3)
root.bind("<Key>", handler2)
root.bind("<Button-1>", handler4)
frame.pack()
frame.focus()

root.mainloop()
```

Now that the first handler has been modified to return "break," you can see that the "o" and "k" keystroke events no longer propagate to the root window, so each keystroke is reported *either* by the Frame or by the root window.

Mouse clicks work similarly, though in those cases *some* clicks are reported by both widgets. Clicks in the lower-right quadrant of the frame don't propagate to the root window, because their x and y attributes are both greater than 50.

Adding Menus to Your Programs

Computer users are used to seeing a menu bar at the top of a program's window. Each word on the bar will drop down a list of menu choices of varying lengths when clicked. (Ellipse's **Search** menu, for example, contains three items).

Building a Menu Bar

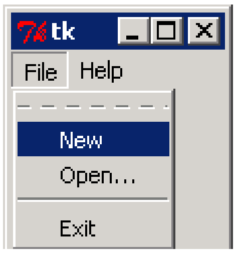
To add a menu bar to a window, instantiate a Menu widget with the window as its parent, and configure it as the window's **menu** item. Then you can add a pulldown Menu widget to the window's menu bar using the menu bar as its master and calling its **add_cascade()** method. Finally, you add choices to the pulldown using the pulldown's **add_command()** method.

Let's try it. In your **MoreGUI/src** folder, create a program named **menudemo.py** as shown:

CODE TO TYPE:

```
from tkinter import *

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.configure(height=75, width=75)
        # create a menu bar
        menu = Menu(root)
        root.config(menu=menu)

        filemenu = Menu(menu)
        menu.add_cascade(label="File", menu=filemenu)
        filemenu.add_command(label="New", command=self.callback1)
        filemenu.add_command(label="Open...", command=self.callback2)
        filemenu.add_separator()
        filemenu.add_command(label="Exit", command=self.callback3)

        helpmenu = Menu(menu)
        menu.add_cascade(label="Help", menu=helpmenu)
        helpmenu.add_command(label="About...", command=self.callback4)

        self.pack()

    def callback1(self):
        print("You selected 'File | New'")

    def callback2(self):
        print("You selected 'File | Open...'")

    def callback3(self):
        print("You selected 'File | Exit'")
        self.quit()

    def callback4(self):
        print("You selected 'Help | About...'")

root = Tk()
app = Application(master=root)
app.mainloop()
```

Run the program; you'll see a window with two items on its menu bar, like the one shown below. Each menu item prints out its identifying information, and the **File | Exit** item actually terminates the program by calling the frame's **quit()** method.



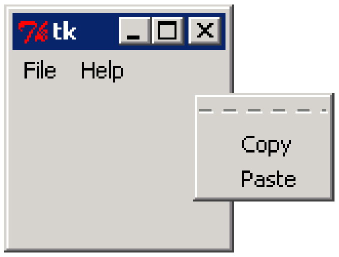
Creating Popup Menus

You can also create menu structures that display on demand. The usual stimulus for display of a so-called "context menu" is a right-click. So you can bind a **<Button-3>** event to the widget you want to provide the menu, and then call the menu's **post()** method to display it from the right-button event handler. You can extract the screen coordinates of the cursor from the event passed to the handler to make the menu display at the current cursor position. Let's give that a try. Modify **menudemo.py** as shown:

CODE TO TYPE:

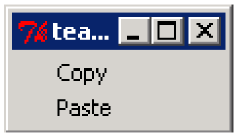
```
from tkinter import *

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.configure(height=75, width=75)
        # create a menu bar
        menu = Menu(root)
        root.config(menu=menu)

        filemenu = Menu(menu)
        menu.add_cascade(label="File", menu=filemenu)
        filemenu.add_command(label="New", command=self.callback1)
        filemenu.add_command(label="Open...", command=self.callback2)
        filemenu.add_separator()
        filemenu.add_command(label="Exit", command=self.callback3)

        helpmenu = Menu(menu)
        menu.add_cascade(label="Help", menu=helpmenu)
        helpmenu.add_command(label="About...", command=self.callback4)

        self.cmenu = Menu(self)
        self.cmenu.add_command(label="Copy", command=self.copy)
        self.cmenu.add_command(label="Paste", command=self.paste)
        self.bind("<Button-3>", self.popup)

        self.pack()

    def callback1(self):
        print("You selected 'File|New'")

    def callback2(self):
        print("You selected 'File|Open...'")

    def callback3(self):
        print("You selected 'File|Exit'")
        self.quit()

    def callback4(self):
        print("You selected 'Help|About...'")

    def copy(self):
        print("Context command 'Copy' selected")

    def paste(self):
        print("Context command 'Paste' selected")

    def popup(self, event):
        self.cmenu.post(event.x_root, event.y_root)

root = Tk()
app = Application(master=root)
app.mainloop()
```

Click the right mouse button (or if you're using a left-handed mouse, click the left button) inside the program's frame; the context menu appears at the position where you clicked:



Tkinter Tearoff Menus

You may be wondering why menus include a dotted line across the top of them. This is a non-standard convenience feature of **tkinter** menus: if you click the dotted line, the menu becomes a separate window (which usually appears at the top-left of your screen) and you can make selections from the window. Below, you see the context menu from the example above, rendered as a separate window. Clicking on the selections works just as if you had brought up the menu using the right button:



If you don't want this feature to be active in your windows, add the **tearoff=False** argument to the menu creation call. That way your users won't see a feature they may not understand.

Dialog Boxes

Creating Simple Dialogs

The class of windows called dialog boxes share many characteristics. They are usually *modal*, which is to say the program behind them becomes non-responsive until the dialog box is either completed or canceled, and they are typically only displayed when a specific task needs to be performed.

Dialogs aren't usually designed to be resized, and are often laid out with the grid manager to accommodate regular rows of labeled entry fields. Tkinter provides a **simpledialog** module that defines a **dialog** class that you can subclass to define your own dialogs.

The **dialog** class provides a basis for extension, including two buttons to complete or cancel the dialog. As an example of **dialog**, we'll use a program that subclasses the **dialog** class to provide an indication of whether the dialog was completed or canceled by adding a **result** attribute.

When painted, a subclass of **simpledialog.Dialog** will contain two buttons: **OK** and **Cancel**. The subclass provides a **body(self, master)** method. This method creates widgets that are children of the **master** argument. It also provides an **apply(self)** method, which will be called only if the **OK** button is clicked.

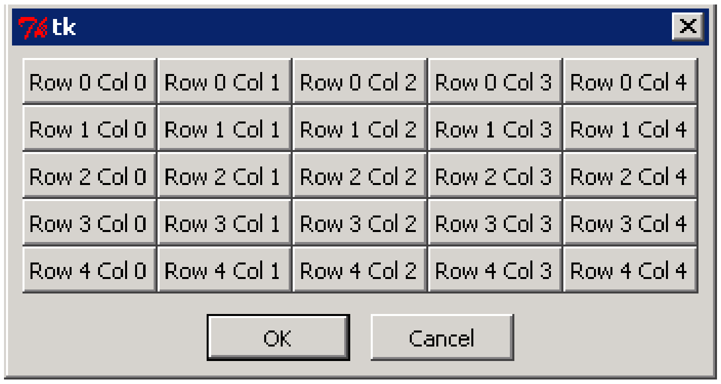
The **body()** method sets a result attribute to a default value that indicates the dialog was canceled. Then the **apply()** method sets an indication that the **OK** button was clicked. The dialog is modal, which means that the main program will not be given control until the user dismisses the dialog. This only happens when the user clicks **OK** or **Cancel**. The code that creates the dialog can look at the result immediately afterwards, and determine whether the dialog should be considered valid.

Enter the code below as **dialog.py** in the **MoreGUI/src** folder:

CODE TO TYPE:

```
from tkinter import *
from tkinter.simpledialog import Dialog

class MyDialog(Dialog):
    def body(self, master):
        self.result = None
        for r in range(5):
            for c in range(5):
                b = Button(master, text="Row {0} Col {1}".format(r, c))
                b.grid(row=r, column=c)
        print("Dialog created")

    def apply(self):
        self.result = "OK"

class Application(Frame):
    def create_dialog(self):
        d = MyDialog(self)
        print(d.result)

    def create_widgets(self):
        self.d_button = Button(self, text="Dialog...", command=self.create_dialo
g)
        self.d_button.pack({"side": "left"})

        self.QUIT = Button(self, text="Quit", fg="red", command=self.quit)
        self.QUIT.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.grid()
        self.create_widgets()

root = Tk()
app = Application(master=root)
app.mainloop()
```

Save and run it. Click **Dialog...**. You'll see a window like this:

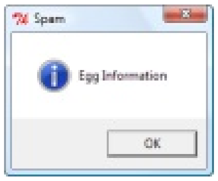
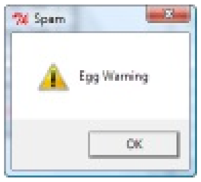
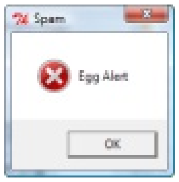
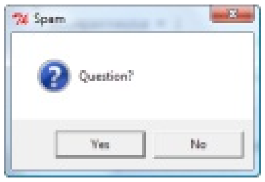
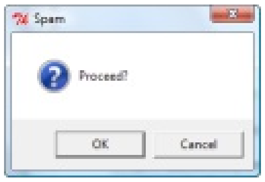
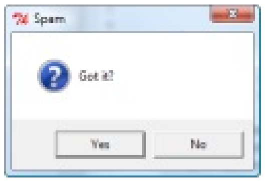
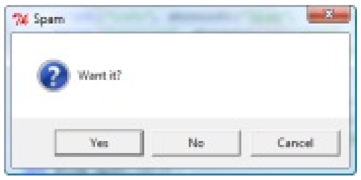
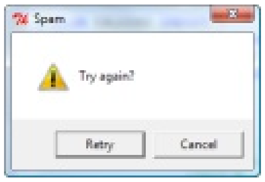


You can resize the dialog, but its contents don't respond to this activity. If you click **OK**, the **apply()** method is called and "OK" prints in the console. If you click **Cancel**, "None" is printed.

Some Ready-Made Dialogs

tkinter provides a number of dialog boxes already programmed for specific purposes. The first set is imported from the **tkinter.messagebox** module. They all take a **title** and a **message** argument, and you can follow those with further keyword arguments to tailor their appearance and behavior:

Dialog Name	Appearance

showinfo	 Egg Information
showwarning	 Egg Warning
showerror	 Egg Alert
askquestion	 Question?
askokcancel	 Proceed?
askyesno	 Got it?
askyesnocancel	 Want it?
askretrycancel	 Try again?

The keyword arguments available to use include **default**, which specifies the button selected if the user presses **Enter**. The button options are: ABORT, RETRY, IGNORE, OK, CANCEL, YES, or NO. These constants are defined in the **tkinter.messagebox** module along with the dialogs.

You can also set the **icon** keyword argument to ERROR, INFO, QUESTION, or WARNING, depending on which graphic you want to include with the message. You can set the **type** argument to be: ABORTRETRYIGNORE, OK, OKCANCEL, RETRYCANCEL, YESNO, or YESNOCANCEL.

The **askcolor** dialog, from the **tkinter.colorchooser** module, allows you to tell your programs the color you want something to be. It normally returns a two-element tuple; the first element is a tuple of RGB values, the second is a string representing the color format used for web content (#RRGGBB). If you cancel the selection, both elements of the tuple are **None**.

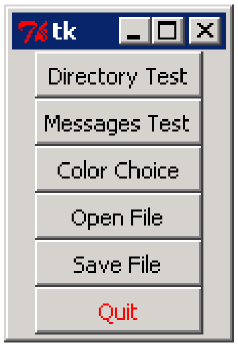
The **filedialog** module provides support for selecting directories and files. With files, **filedialog** supports either loading (providing the selected file exist) or saving. With modules, dialogs will display tkinter's limitations. We'll see **filedialog** in action in our last example of this lesson. In your **MoreGUI/src** folder, create **dialogdemo.py** as shown:

CODE TO TYPE:

```
from tkinter import *
from tkinter.filedialog import LoadFileDialog, SaveFileDialog, Directory
from tkinter.colorchooser import askcolor
from tkinter.messagebox import (showinfo, showwarning, showerror, askquestion,
                                askokcancel, askyesno, askyesnocancel, askretryc
ancel)


class Application(Frame):
    def askdir(self):
        d = Directory(self)
        print(d.show())

    def messages(self):
        print("info", showinfo("Spam", "Egg Information"))
        print("warning", showwarning("Spam", "Egg Warning"))
        print("error", showerror("Spam", "Egg Alert"))
        print("question", askquestion("Spam", "Question?"))
        print("proceed", askokcancel("Spam", "Proceed?"))
        print("yes/no", askyesno("Spam", "Got it?"))
        print("yes/no/cancel", askyesnocancel("Spam", "Want it?"))
        print("try again", askretrycancel("Spam", "Try again?"))

    def file_open(self):
        d = LoadFileDialog(self)
        fname = d.go("nosuch.txt", "*.py")
        if fname is None:
            print("Canceled...")
        else:
            print("Open file", fname)

    def file_save(self):
        d = SaveFileDialog(self)
        fname = d.go("example", "*.py")
        if fname is None:
            print("Canceled...")
        else:
            print("Saving file", fname)
    def color(self):
        d = askcolor()
        print(d)

    def createWidgets(self):
        d_button = Button(self)
        d_button.config(width=12, text="Directory Test", command=self.askdir)
        d_button.pack(side=TOP)
        m_button = Button(self)
        m_button.config(width=12, text="Messages Test", command=self.messages)
        m_button.pack()
        c_button = Button(self)
        c_button.config(width=12, text="Color Choice", command=self.color)
        c_button.pack()
        l_button = Button(self)
        l_button.config(width=12, text="Open File", command=self.file_open)
        l_button.pack()
        s_button = Button(self)
        s_button.config(width=12, text="Save File", command=self.file_save)
        s_button.pack()
        self.QUIT = Button(self)
        self.QUIT.config(width=12, text="Quit", fg="red", command=self.quit)
        self.QUIT.pack(side=TOP)

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()
```

```
root = Tk()
app = Application(master=root)
app.mainloop()
```

Save and run it. You'll see a window with various buttons:



Click the buttons for examples of the **filedialog** uses we talked about.

And there you have it! This concludes our discussion of the **tkinter** library. Next up—Databases! See you there!

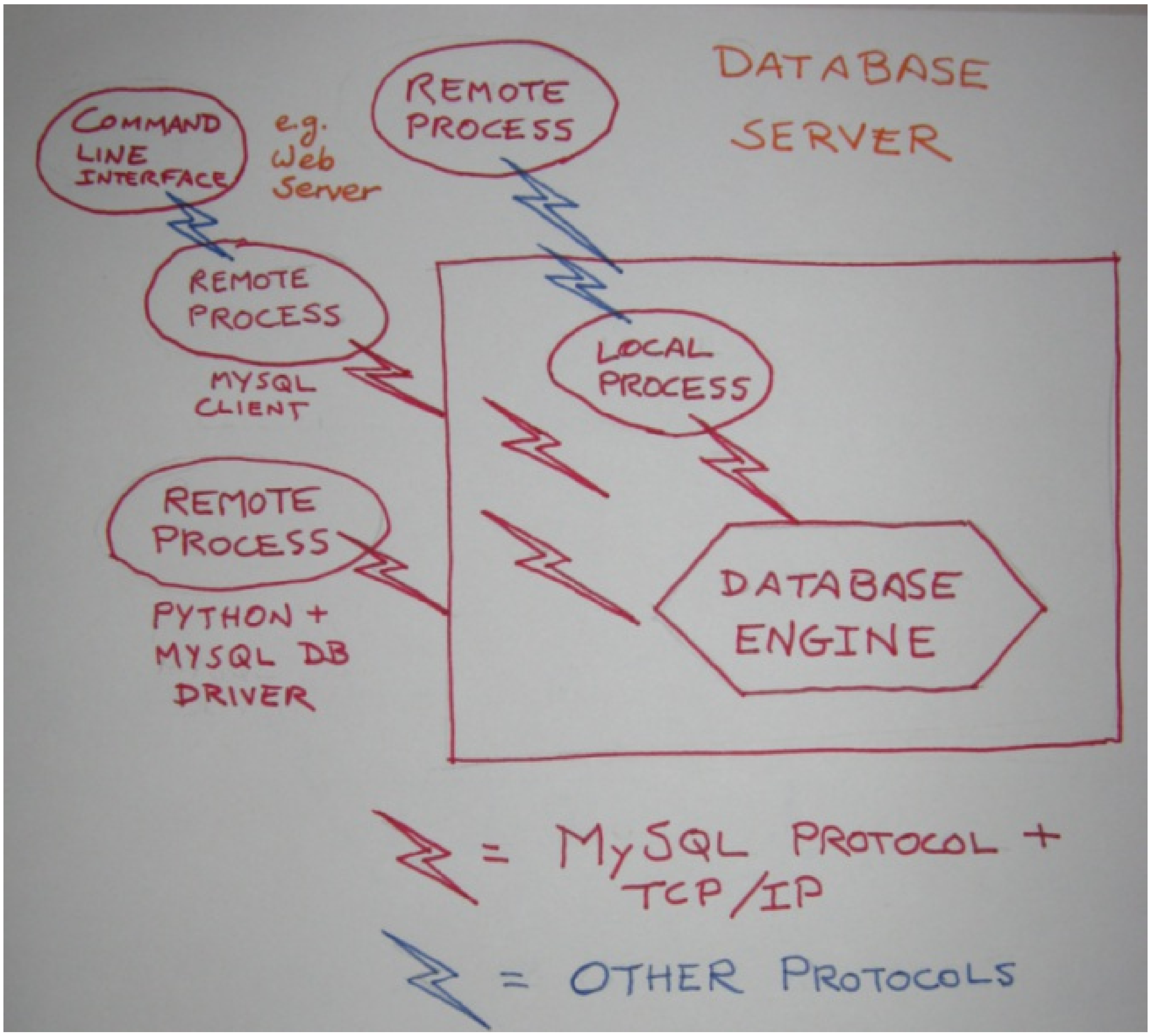
Handling Databases

Relational Databases: Representing the World in Tables

You haven't had enough yet? Good! Let's keep our momentum going then, and start talking about relational databases. Relational databases are based on complex discrete mathematics. Fortunately though, we don't need to master all of those complex mathematics in order to get the most from a database: the concepts are actually pretty intuitive.

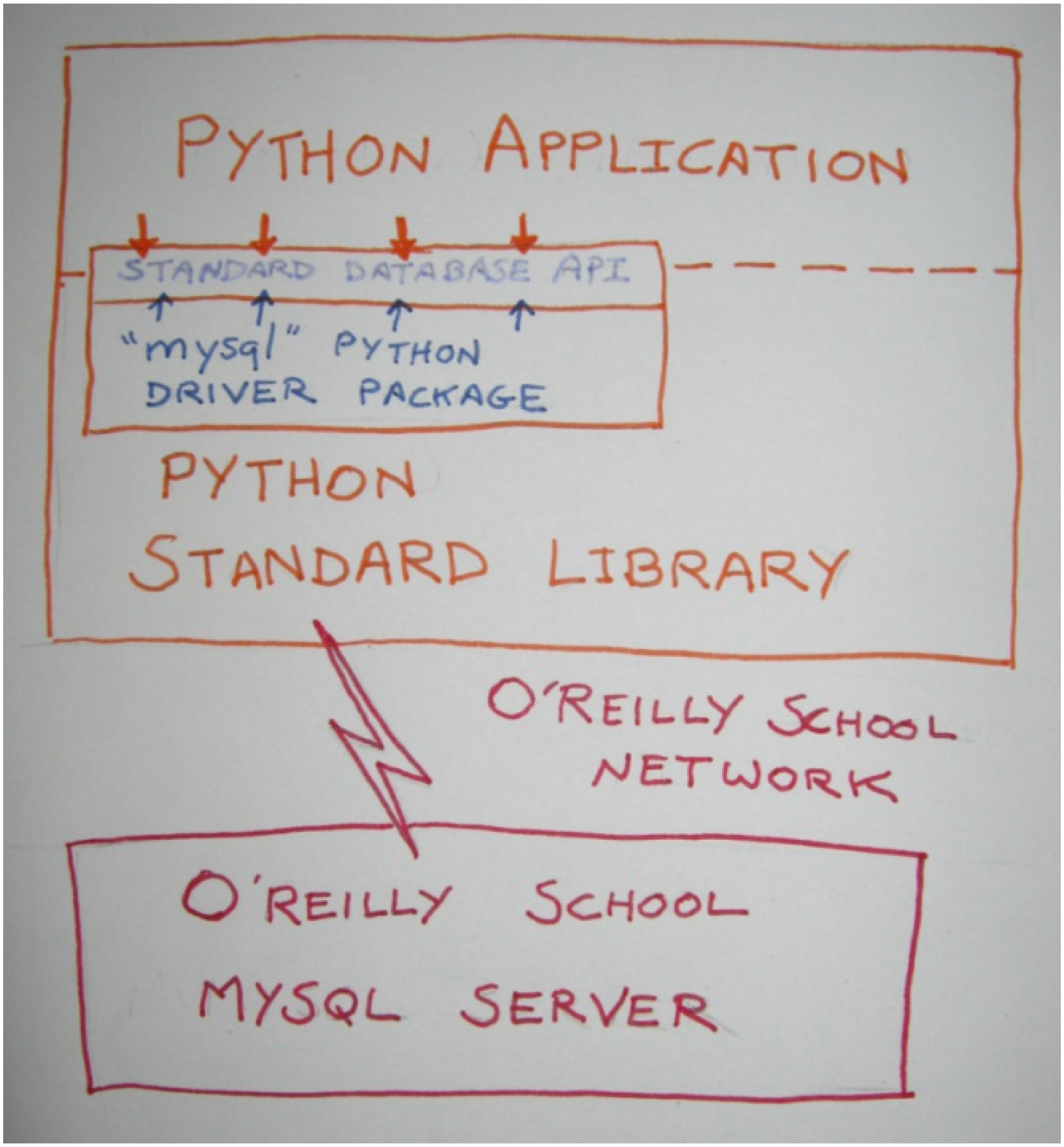
Relational databases use a language called the *Structured Query Language* (abbreviated as *SQL*) to define data structures, and store and retrieve information. SQL is different from most computer languages in that it is *declarative*—you don't tell the database how to produce what you want, you simply describe what you want it to do and the database works out how best to do it.

Database systems are often built as "client/server" systems—your program is a client (maybe one of many) of a server program that runs as a separate process, or maybe even on an entirely separate "database server" computer. The diagram below of a MySQL database environment, depicts how some programs use the *MySQL protocol*—that's a way of speaking SQL that's particular to MySQL clients and servers—to the database server:



Okay, let's get this party started. First we'll run SQL on a MySQL database server elsewhere in the O'Reilly School network. Your Ellipse setup will be the client in the top left-hand corner, connecting via the *SSH* protocol to another computer, on which you will enter SQL commands to a Linux MySQL client program. The database client program will then present your SQL to the database server, where the engine will process it.

Then we'll move on to Python database programming. Your programs will resemble the lower of the two remote processes in the diagram, where you use Python to interact with the database, by means of a special piece of driver software. Think of your program like this:



(Do you love our sweet diagrams or what?) In this course, you will use a MySQL database provided by OST. Your database uses the same credentials (login name and password) as the other O'Reilly School systems you use, and its name is the same as your user name. We've saved you the trouble of installing your own database server; the care and feeding of these beasts requires some expertise! Your database programs will maintain their data on O'Reilly's servers.

MySQL conforms closely to the Python DBAPI specifications for the way your programs should interact with the database. Support for many relational databases, both open source and proprietary, is readily available using well-tried third-party modules.

SQL is the common interface between databases and their applications. This means that you don't need to incorporate information about the physical representation of data on the storage media. Also, SQL processors do all the "grunt work" of optimization, and do not require programmers to specify the complex operations that complex queries require.

Your First Database Interactions

Access to a Database

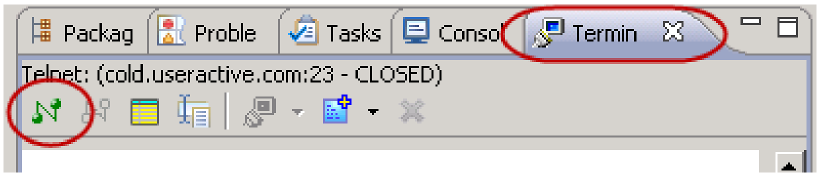
The MySQL database server is a popular open-source database software, and is available for use on a wide variety of platforms. We'll use the MySQL database for our examples.

As we mentioned earlier, in this course, you have access to the MySQL database on the OST server. You can access that database using the same username and password that you use to log onto your courses. Your username is also the name of your database.

If you write in an SQL dialect that most database systems support, your SQL (and therefore your programs) should run successfully against most database systems without change. Even though SQL is standardized, each database vendor has a different interface, as well as different extensions of SQL. We create our notes and examples here to be general-purpose, so you can use them on this or other databases, but remember that changes *may* be required.

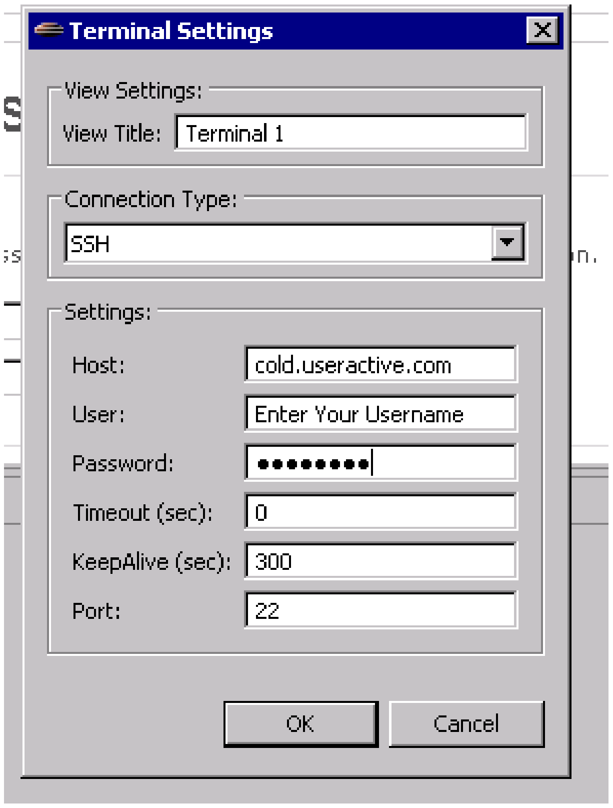
First, we'll connect to your Unix account using SSH. To do that, use the **Terminal** tab in Ellipse.

Click the **Terminal** tab and then the **Connect** icon:



Telnet: (cold.useractive.com:23 - CLOSED)

If a Terminal Settings dialog appears, select SSH for the Connection Type, enter **cold.useractive.com** for the Host, and enter the username and password you used to get into this course, and click **OK**:

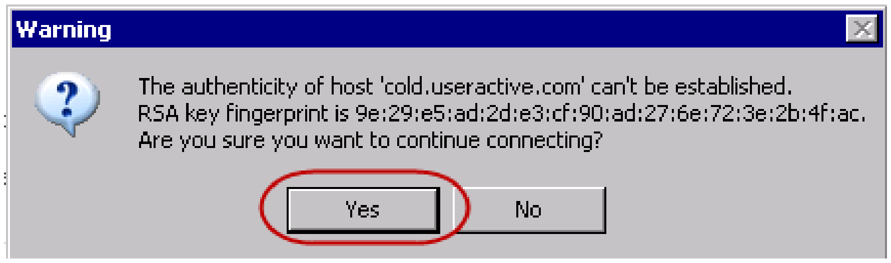


View Settings:
View Title: Terminal 1

Connection Type:
SSH

Settings:
Host: cold.useractive.com
User: Enter Your Username
Password: ●●●●●●●●●
Timeout (sec): 0
KeepAlive (sec): 300
Port: 22

An authenticity warning appears:


Warning

The authenticity of host 'cold.useractive.com' can't be established.
RSA key fingerprint is 9e:29:e5:ad:2d:e3:cf:90:ad:27:6e:72:3e:2b:4f:ac.
Are you sure you want to continue connecting?

Click **Yes** to continue and make the connection.

Running mysql

At the cold:~$ prompt, type:

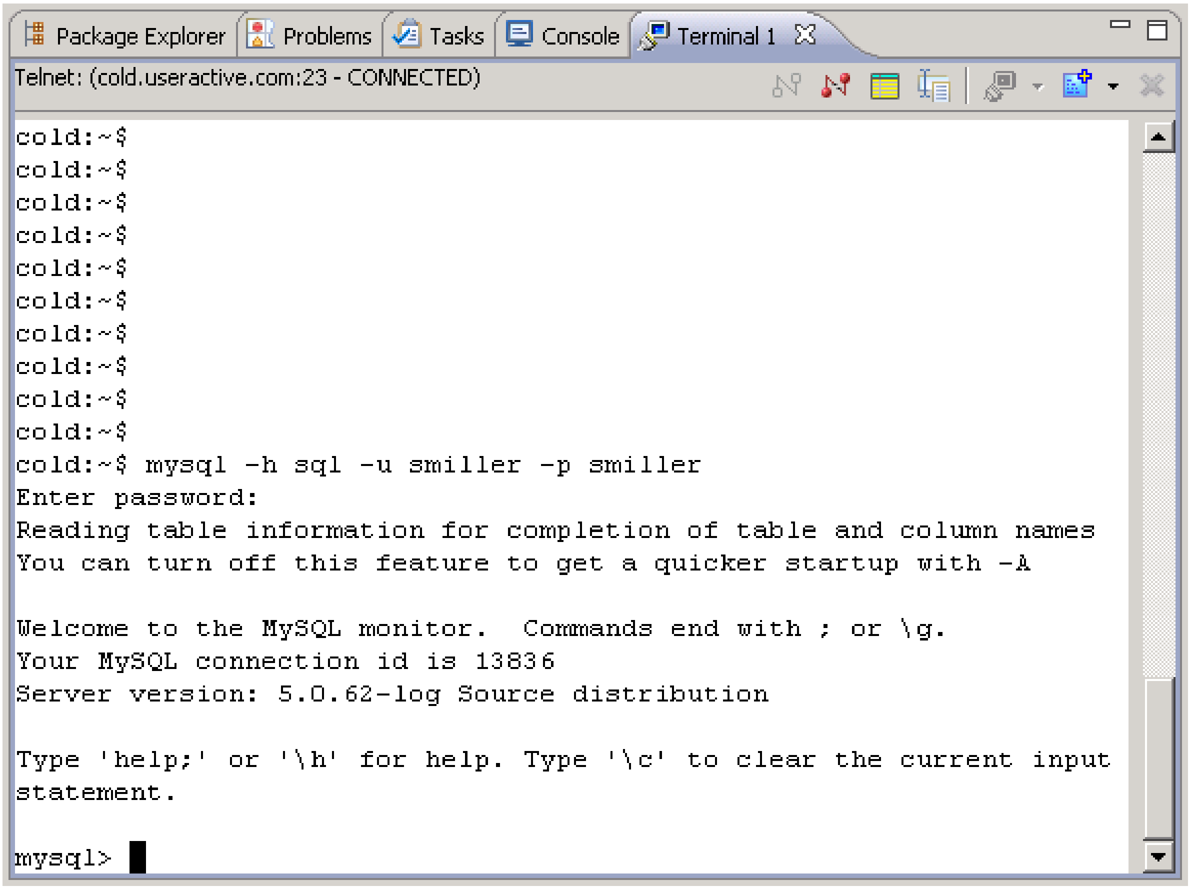
INTERACTIVE SESSION:

```
cold:~$ mysql -h sql -u <username> -p <username>
```

OBSERVE: Starting mysql at the cold Prompt

```
mysql -h sql -u <username> -p <username>
```

(At the first instance of **username**, enter the name you use to login to this course; for the second instance, **username** should be replaced with your login name as well—this will become the name of the database.) When prompted for a password (as required by the -p option), enter the password you use to login to this course. You'll see a **mysql>** prompt:



Telnet: (cold.useractive.com:23 - CONNECTED)

```
cold:~$
cold:~$
cold:~$
cold:~$
cold:~$
cold:~$
cold:~$
cold:~$
cold:~$
cold:~$
cold:~$ mysql -h sql -u smiller -p smiller
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13836
Server version: 5.0.62-log Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql>
```

Excellent! You're in!

Structured Query Language

Structured Query Language (SQL) is a command-based free-form language. All whitespace is considered equivalent, and the keywords, table names, and column names are not case-sensitive. It is common convention to write SQL keywords in upper-case letters so they can be readily identified.

In fact, MySQL is picky about table names, and it's best practice to remain consistent, using the exact table names that you create originally. No database will ever complain because you got the case right!

Each statement in SQL begins with a characteristic verb or phrase, which indicates the broad purpose of the statement. SQL actually includes three sub-languages, two of which we'll consider in this course: Data Definition Language and Data Manipulation Language. (The third, Data Control Language or DCL, is used to determine which users get permission to perform which operations on which pieces of data, and is outside the scope of this course.)

Data Definition Language (DDL)

DDL is the subset of SQL that allows you to define and modify database objects such as tables and indexes. There are three basic verbs used in DDL:

- **CREATE** inserts new definitions into the data dictionary.
- **DROP** removes definitions from the data dictionary.
- **ALTER** modifies definitions already present in the data dictionary.

Here is a description of two relational tables that might be part of a library information system. At the mysql prompt, enter the SQL shown below to create two tables:

INTERACTIVE SESSION:

```
mysql> CREATE TABLE Book(
  -> BkISBN CHAR(12) NOT NULL,
    -> BkTitle VARCHAR(30) NOT NULL,
    -> BkPubNo INT,
    -> BkYear INT);
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE Publisher(
  -> PubNo INT PRIMARY KEY,
  -> PubName VARCHAR(25),
  -> PubURL VARCHAR(50));
Query OK, 0 rows affected (0.01 sec)
```

Here you create a table in SQL using the phrase "CREATE TABLE" followed by the name of the table you want to create, followed by a parenthesized list of column specifications separated by commas. Each column specification determines the data type of the values that are stored in the column and can specify other constraints (PRIMARY KEY specifies a constraint on the Publisher table—we'll learn about others later).

In many database systems, constraints do not need to be specified when the table is created. They can be added later using the ALTER TABLE statement. Now enter these statements in mysql for the tables you defined:

INTERACTIVE SESSION:

```
mysql> ALTER TABLE Book
    ->    ADD CONSTRAINT Bk_PK
    ->    PRIMARY KEY(BkISBN);

mysql> ALTER TABLE Book
    ->    ADD CONSTRAINT Bk_Pub_FK
    ->    FOREIGN KEY (BkPubNo) REFERENCES Publisher;
```

The second statement expresses the fact that a *relationship* exists between Book and Publisher: each book is related to (published by) one of the publishers in the Publisher table. A publisher is identified by its primary key (the value of the **PubNo** column). Consequently, the publisher of a given book is recorded by storing the appropriate **PubNo** value from Publisher, as the value of the **BkPubNo** column for the row representing the book. We'll talk more about relationships later.

Data Manipulation Language (DML)

This is the most commonly used subset of SQL; it is used to manipulate and query the data in the relational structures maintained by the DDL, and is what updates the model and answers questions based on its content. There are four statements in the Data Manipulation Language:

- **INSERT** adds new rows to user tables.
- **SELECT** retrieves information from one or more tables in the database (including data dictionary tables if requested).
- **UPDATE** allows changes to be made to existing rows in user tables.
- **DELETE** removes rows from user tables.

INSERT: Adding A Row to a Table

Now, we'll use the INSERT statement to insert some book and publisher data. In mysql, enter the code as shown:

INTERACTIVE SESSION:

```
mysql> INSERT INTO Publisher (PubNo, PubName, PubURL)
    -> VALUES (1, 'O''Reilly', 'www.ora.com');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Publisher (PubNo, PubName, PubURL)
    -> VALUES (2, 'New Riders', 'www.newriders.com');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Book (BkISBN, BkTitle, BkPubNo, BkYear)
    -> VALUES('7807', 'Python Web Programming', 2, 2002);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Book (BkISBN, BkTitle, BkPubNo, BkYear)
    -> VALUES('0596', 'Learning Python', 1, 2009);
Query OK, 1 row affected (0.00 sec)
```

Each INSERT statement adds one row to the database table, named after the "INSERT INTO" clause. The values in the VALUES list match up with the columns given in the list immediately following the table name.

Note You may see code containing SQL INSERT statements that don't include the list of column names, instead relying on the order of the column names when the table was created. This is not a best practice.

SELECT: Retrieve Data from One or More Tables

Once you have data in your database, you can retrieve information using the SELECT statement. Enter the code below as shown:

```
INTERACTIVE SESSION:

mysql> SELECT BkTitle, BkISBN, PubName
    -> FROM Book JOIN Publisher ON BkPubNo = PubNo;
+------------------------+--------+------------+
| BkTitle                | BkISBN | PubName    |
+------------------------+--------+------------+
| Python Web Programming | 7807   | New Riders |
| Learning Python        | 0596   | O'Reilly   |
+------------------------+--------+------------+
2 rows in set (0.05 sec)

mysql>
```

The statement above retrieves the ISBN and title of the book from the Book table and the relevant publisher's name from the Publisher table, and puts them together—this is called *joining* the tables. This results in the two rows of data shown.

UPDATE: Modify Existing Data in a Table

The UPDATE statement is used to modify existing data, and can change zero, one, or more rows in a single table. Suppose a second edition of *Python Web Programming* were published, then the database could be modified to reflect the new book Add the code below as shown:

```
INTERACTIVE SESSION:

mysql> UPDATE Book SET BkTitle='Python Web Programming, 2nd Ed',
    -> BkYear=2010
    -> WHERE BkISBN='7807';
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>

mysql> SELECT * FROM Book;
+--------+--------------------------------+---------+--------+
| BkISBN | BkTitle                        | BkPubNo | BkYear |
+--------+--------------------------------+---------+--------+
| 7807   | Python Web Programming, 2nd Ed |       2 |   2010 |
| 0596   | Learning Python                |       1 |   2009 |
+--------+--------------------------------+---------+--------+
2 rows in set (0.04 sec)

mysql>
```

The output from the SELECT statement (the "*" simply means "all columns") shows that, in this case, precisely one row was updated by the UPDATE statement. That happened because the WHERE clause specified a condition that was only met by one row in the given table.

DELETE: Remove Rows From a Table

The DELETE statement removes all rows meeting a specific condition, again expressed in a WHERE clause. You need to be *really* careful here—if you do not specify a WHERE clause, all rows in the table will disappear! At the MySQL prompt type the code below as shown:

INTERACTIVE SESSION:

```
mysql> DELETE FROM Book WHERE BkISBN='0596';
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM Book;
+--------+--------------------------------+---------+--------+
| BkISBN | BkTitle                        | BkPubNo | BkYear |
+--------+--------------------------------+---------+--------+
| 7807   | Python Web Programming, 2nd Ed |       2 |   2010 |
+--------+--------------------------------+---------+--------+
1 row in set (0.04 sec)

mysql>
```

Having No Data: The Null Value

Relational systems use a special value called *null* to represent the fact that either no data is available for a specific column in a given row, or that that column is irrelevant in the case of the particular row in question.

While the null value (indicated as NULL in SQL statements) has its uses, you need to be careful of its counter-intuitive properties. Because the null value in effect represents the absence of data, it introduces a third possibility beyond true or false—unknown—as the result of a comparison.

Consequently, if you are testing a column for a given value, NULLs will not be included whether you test for equality or inequality (which you do IN SQL using "=" and "<>", respectively). Since we are used to the value of a comparison being true or false, it's easy to forget to take this oddity into account. Use mysql to check this out. Enter this SQL to see how NULL values affect comparisons:

INTERACTIVE SESSION:

```
mysql> INSERT INTO Book (BkISBN, BkTitle, BkPubNo)
    -> VALUES ('1234', 'Pythonic Attitudes', 2);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO Book (BkISBN, BkTitle, BkPubNo, BkYear)
    -> VALUES ('0987', 'My Little Python', 1, 2005);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM Book;
+--------+--------------------------------+---------+--------+
| BkISBN | BkTitle                        | BkPubNo | BkYear |
+--------+--------------------------------+---------+--------+
| 7807   | Python Web Programming, 2nd Ed |       2 |   2010 |
| 1234   | Pythonic Attitudes             |       2 |   NULL |
| 0987   | My Little Python               |       1 |   2005 |
+--------+--------------------------------+---------+--------+
3 rows in set (0.05 sec)

mysql>
```

See the NULL value in the BkYear column for "Pythonic Attitudes?" (That's right, I said it. Pythonic Attitudes.) Now we'll run a number of queries on this updated data. Enter this SQL in the MySql Terminal Window:

INTERACTIVE SESSION:

```
mysql> SELECT COUNT(*) FROM Book;
+----------+
| COUNT(*) |
+----------+
|        3 |
+----------+
1 row in set (0.04 sec)

mysql> SELECT COUNT(*) FROM Book WHERE BkYear <= 2005;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.04 sec)

mysql> SELECT COUNT(*) FROM Book WHERE BkYear > 2005;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.04 sec)

mysql>
```

So, we have three books; one published 2005 or earlier, and one published after 2005, and...wait a minute! If you don't see the problem here, think about how many books there are in the Books table. The first query answers that: there are three. Okay, so how many of them were published in or before 2005? The second query tells us there is one. The third query tells us that there is only one book published *after 2005*—the answer is again, one. But one plus one isn't three, and it's easy to overlook that the book with no year data couldn't be included in either result set.

You need to be careful of little things like this when your data allows NULL values, and we recommend that another best practice is to allow NULL values only where you actively want to permit them. So, you find out how many books there are in total by giving the SQL **COUNT()** function a "*" (all rows) argument. Counting the **BkYear** column is no good—because NULL values in that column are omitted from the **COUNT()**. Let's verify in the MySQL Terminal Window that NULLs are not COUNTed. Type in the code below as shown:

INTERACTIVE SESSION:

```
mysql> SELECT COUNT(BkYear) FROM Book;
+---------------+
| COUNT(BkYear) |
+---------------+
|             2 |
+---------------+
1 row in set (0.04 sec)

mysql>
```

Only two of our books have a BkYear. When we entered the "Pythonic Attitudes" book data, we didn't include a value for the year (sorry, we did that deliberately). In the result from the **SELECT * FROM Book** query, we mentioned that the value for **BkYear** for that row was **NULL**.

Note Data that can take the NULL value is sometimes referred to as *optional*.

So, we used **COUNT()**, a SQL function that aggregates the number of rows that meet the given condition. A count of all rows clearly shows three rows, but we only saw one row with a year less than or equal to 2005 and one with a year that was greater than 2005. The third row didn't get counted by either of those conditions.

The final query, where we explicitly counted the number of **BkYear** entries, makes it apparent that only two rows have an entry in that column.

Creating a Table and Inserting Data

Tables are at the heart of a database. Each table in a properly designed database, holds data concerning precisely one type of thing (an *entity type*, as it is more formally called in the database world). Suppose you want to keep information about a zoo; you would certainly want to record information about the animals, and you could do so in a single table.

Let's create a new database with a table for storing basic information about zoo animals. Again, if it's not still running, open a terminal window and start mysql. Enter the code below in an interactive window to create a new table:

INTERACTIVE SESSION:

```
mysql> CREATE TABLE animal(
    -> id INTEGER PRIMARY KEY AUTO_INCREMENT,
    -> name VARCHAR(50),
    -> family VARCHAR(50),
    -> weight INTEGER);
Query OK, 0 rows affected (0.03 sec)
```

Attributes are Columns, Occurrences are Rows

In the example above, you used a "CREATE TABLE" statement to create a table interactively with four columns: **id**, **name**, **family**, and **weight**. **id** and **weight** hold integer values; **name** and **family** hold character strings of varying length. Each column represents a different piece of data about each animal that can be stored—they are sometimes referred to as "attributes" of the animal entity.

The PRIMARY KEY column designation plays a special role in the table, which we'll discuss at length later. For now, be aware that using its primary key is the only way to guarantee that you are referring to a single row in the table; primary key values are always unique.

Now that we've created our table, let's put some data into it. The code below shows how to use the SQL "INSERT" statement. If you were writing SQL for direct execution by the database, you would write something like this:

INSERT INTO animal (id, name, family, weight) VALUES (1, 'Ellie', 'Elephant', 2350)

Note that SQL *always* uses single quotation marks to delimit string values. In a program though, you usually have the data in variables. While you *could* build the exact SQL statement you want to run using string manipulation, this is a really bad idea. If the data strings are from user input, it is too easy to allow the user to mess up your SQL, sometimes with disastrous results (try searching the web for "SQL injection vulnerability" to see how bad this can be). Fortunately, Python provides a mechanism to avoid these unpleasant security vulnerabilities. Which brings us to consider how your programs will interact with the database.

The Python Database API

We've used SQL at the command line after securely logging in to a remote Linux system. Now we need to learn how to use it from inside of our Python programs, so that instead of just displaying the data we retrieve, we can execute Python statements using the data. That should make things a bit more interesting!

In order for a Python program to be able to talk to a database, it uses a special driver module. We are using the **mysql.connector** module, calling its **Connect()** function to identify ourselves and obtain a database connection. Once the connection is created, a *cursor* is used to execute SQL commands over that connection. This program only inserts data into the database; it does not attempt to retrieve any data.

The DBAPI provides a solution to the SQL injection problem by allowing you to make what are called "parameterized queries." They're a little like passing arguments to Python functions. You include a special "parameter mark" ("%s" for the **mysql.connector** module we use in this course) in the SQL statement to represent each piece of data, and then provide the data itself as an additional tuple to your database cursor's **execute()** method. Let's write a program to insert data into our animal table. In the **HandlingDatabases/src** folder, create **tablepop.py** as shown:

CODE TO TYPE:

```
"""
Populates a table with data from a Python tuple.
"""
import mysql.connector
from database import login_info

if __name__ =="__main__":

    db = mysql.connector.Connect(**login_info)
    cursor = db.cursor()

    data = (
            ("Ellie", "Elephant", 2350),
            ("Gerald", "Gnu", 1400),
            ("Gerald", "Giraffe", 940),
            ("Leonard", "Leopard", 280),
            ("Sam", "Snake", 24),
            ("Steve", "Snake", 35),
            ("Zorro", "Zebra", 340)
            )

    cursor.execute("DELETE FROM animal")
    for t in data:
        cursor.execute("""
        INSERT INTO animal (name, family, weight)
        VALUES (%s, %s, %s)""", t)

    db.commit()
    print("Finished")
```

When you save and run this program, naturally, it raises an exception; rather than insert our login credentials into this program, we're importing them as something called **login_info** from a module named **database**, which doesn't yet exist.

Note You may see a warning on the **import mysql.connector** line in this program. Ignore it for now.

Create **database.py** in your **HandlingDatabases/src** folder, entering *your* login username in place of "*username*" and your password in place of "*password*":

CODE TO TYPE:

```

USERNAME = "username"
PASSWORD = "password"

login_info = {
            'host': "sql.oreillyschool.com",
            'user': USERNAME,
            'password': PASSWORD,
            'database': USERNAME,
            'port': 3306
            }
```

This code creates a dict. The dict's items will become keyword arguments to the **mysql.connector.Connect()** function (remember, the "**" tells the interpreter to convert the dict into a set of keyword arguments). Normally, to connect to a database server, you need to know a few pieces of information, which you have to pass to the driver when connecting to the database. The names of the first four arguments (host, user, password, and database) will probably make their purpose obvious. The fifth argument, port, is required so the driver knows exactly where to connect on the database server. Save this **database** module, then re-run tablepop.py. The program inserts seven rows into your database's animal table, and prints FINISHED. But don't take our word for it—check for yourself after you learn what your program did! Let's look at the code more closely:

OBSERVE: tablepop.py

```
"""
Populates a table with data from a Python tuple.
"""
import mysql.connector
from database import login_info

if __name__ =="__main__":

    db = mysql.connector.Connect(**login_info)
    cursor = db.cursor()

    data = (
            ("Ellie", "Elephant", 2350),
            ("Gerald", "Gnu", 1400),
            ("Gerald", "Giraffe", 940),
            ("Leonard", "Leopard", 280),
            ("Sam", "Snake", 24),
            ("Steve", "Snake", 35),
            ("Zorro", "Zebra", 340)
            )

    cursor.execute("DELETE FROM animal")
    for t in data:
        cursor.execute("""
        INSERT INTO animal (name, family, weight)
        VALUES (%s, %s, %s)""", t)

    db.commit()
    print("Finished")
```

First let's consider the database connection. The statement **db = mysql.connector.Connect(**login_info)** is equivalent to **db = mysql.connector.Connect(host= ..., user= ..., ...)**, with the dict imported from the **database** module providing both the names and the values of the parameters to the **Connect()** function.

Next, the statement **cursor = db.cursor()** creates a database cursor, which is how we present SQL statements to the database for execution (you can create several cursors on the same connection if you want to, but usually you won't do that). Next, the program loops over each of the tuples in **data**, presenting each tuple as the second argument to a call to the cursor's **execute()** method.

Each time we called the **cursor.execute()** method, we provided the same parameterized SQL INSERT statement (containing three "%s" *parameter marks* to indicate where the data should go) as the first argument. The second argument was the tuple of data items. This inserted a new row into the animal table.

Note

The INSERT statement didn't provide a value for the **id** column. Where did the IDs come from? When we created the table, we declared **id** as **INTEGER PRIMARY KEY AUTO_INCREMENT**. AUTO_INCREMENT specifies that the **id** column of any row that is inserted into the table with no **id** value specified, will be set to one greater than the highest value that was ever stored in that column. In practice, this normally means that values start at one and go up, which is what we see here. If you have inserted and deleted other rows (good for you for experimenting!), you might see different numbering.

So, now that we know what was supposed to happen inside **tablepop.py**, we should check to make sure that it ran correctly!

In the Terminal tab, open a connection to the database, and verify the contents of the **animal** table. Type this code at the mysql prompt:

INTERACTIVE SESSION:

```
mysql> SELECT * FROM animal;
+----+---------+----------+--------+
| id | name    | family   | weight |
+----+---------+----------+--------+
|  1 | Ellie   | Elephant |   2350 |
|  2 | Gerald  | Gnu      |   1400 |
|  3 | Gerald  | Giraffe  |    940 |
|  4 | Leonard | Leopard  |    280 |
|  5 | Sam     | Snake    |     24 |
|  6 | Steve   | Snake    |     35 |
|  7 | Zorro   | Zebra    |    340 |
+----+---------+----------+--------+
7 rows in set (0.04 sec)
```

Relationships and Foreign Keys: Referring to Occurrences

You already saw in the book/publisher example that it is possible for a row in one table to refer to a row in another table. Each book indicates its publisher in a column called **BkPubNo** that holds the value of the **PubNo** field of one of the rows in the Publisher table.

The **BkPubNo** column in the Book table is called a *foreign key*—it stores a primary key value from some row in another table. Since primary key values are guaranteed to be unique, a foreign key value refers just once to a single instance of the related entity.

Foreign keys are used to express the fact that relationships exist between two entities. In this case, we might say that "book is-published-by publisher," or equivalently that "publisher publishes book." Since each book can have only one publisher, but any given publisher can publish many books, we say that the relationship is "many-to-one" between book and publisher, or equivalently that it is "one-to-many" between publisher and book.

Relationships can turn mere data into information. Without the relationships, we could not show the publisher of each book in the query we ran earlier.

Integrity Constraints

For data to be stored in the database, it must meet certain rules, which are we'll summarize in a minute. Most relational databases will enforce these rules automatically to maintain the integrity of the relational structures. For this reason, the rules are often referred to as *integrity constraints*. Your application may also have its own integrity requirements imposed on the database content.

For example, it's fairly common in order processing systems to assign each customer a credit limit, to allow them to purchase a certain amount without advance payment. Generally a customer's credit limit will be increased as they demonstrate their trustworthiness. When a new order is received, the system checks how much the customer already owes, and if the new order would take them over their credit limit, it refuses to release the new order (at least without some manual override action). So the constraint there is that each customer's unpaid order total must be less than their credit limit. Constraints imposed because of the organization's requirements are often referred to as *business rules*, or *semantic integrity constraints*, but they are still constraints. They are frequently so complex that it isn't reasonable to expect the database to maintain them without help from code in the application.

In this section, we discuss the integrity constraints that we usually expect the database to maintain without any help.

Primary Keys Identify Occurrences

Each row of a table must be uniquely identifiable. The easiest way to ensure this is to have a column or collection of columns that is guaranteed unique for every row in the table. This column (or collection) is designated as the *primary key*. A primary key that is made up of more than one column is referred to as a *composite primary key*. The database will not allow two rows with the same primary key value to exist. You can see for yourself by trying to create a duplicate **id** value in the animal table. In the interactive console, type the code below as shown:

INTERACTIVE SESSION:

```
mysql> INSERT INTO animal (id, name, family, weight)
    -> VALUES (1, "Harold", "Hyena", 80);
ERROR 1062 (23000): Duplicate entry '1' for key 1
mysql>
```

The error message tells you that the row could not be created because that would have resulted in a duplicate primary key, which in turn would violate the built-in integrity constraint.

No NULLs in Primary Key Values

No part of the primary key may be null. The primary key is used as the unique identifier for the rows of a table. Since the result of a comparison between anything and null is unknown, it would be impossible to answer yes or no to the question "does the primary key of this row have that value?"

No Multi-Valued Attributes

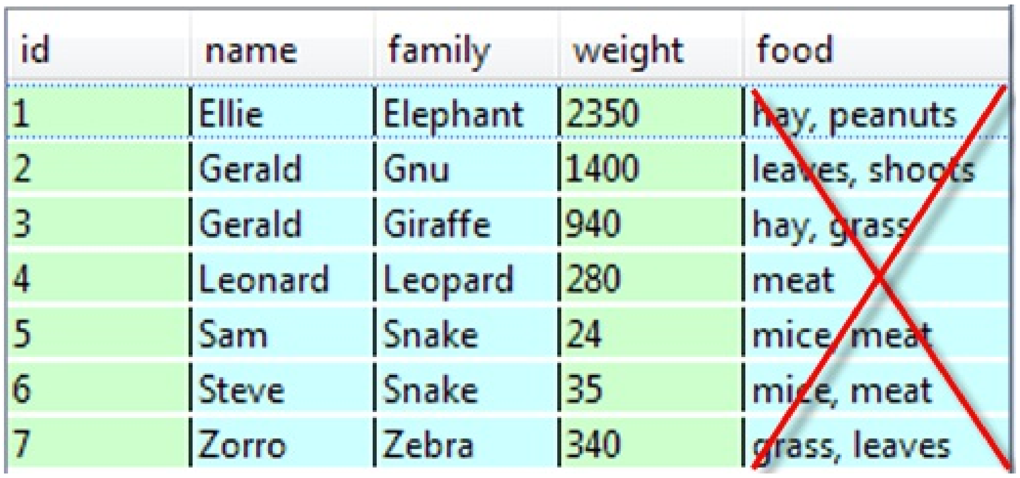
Attribute values must be "atomic". There is no way to store more than one value for a given attribute in any row. If you need to do that, you need to create a relationship instead. You can learn more about this under "Implementing Multi-Valued Attributes" below.

Referential Integrity

Foreign key values must exist as primary key values in the related table. This is a fairly straightforward interpretation of the meaning of relationships. Because a book's publisher is indicated by its **BkPubNo** attribute, the value of that attribute must be a reference to a real publisher.

Implementing Multi-Valued Attributes

What if we wanted to store details about what each animal in our zoo eats? One way to do this would be to add a **food** column to the table. But what if an animal can eat more than one type of food? A common mistake of new database programmers make is to try and store several values in a single column, as in this table:



id	name	family	weight	food
1	Ellie	Elephant	2350	hay, peanuts
2	Gerald	Gnu	1400	leaves, shoots
3	Gerald	Giraffe	940	hay, grass
4	Leonard	Leopard	280	meat
5	Sam	Snake	24	mice, meat
6	Steve	Snake	35	mice, meat
7	Zorro	Zebra	340	grass, leaves

Don't do this. The red X is there to remind you that this is a terrible idea. Using a table like this would make it next to impossible to answer relatively simple queries like "which animals eat grass?" The solution to this problem is to introduce an entirely new entity to store this information, and put the new entity in a relationship with the animal entity by storing the primary key of the animal as an attribute of the new **food** entity. Let's do that now. We'll use some DDL to create the new table and some DML to add the rows. In the **HandlingDatabases/src** folder, create **addfood.py** as shown:

CODE TO TYPE:

```
"""
Create the food table and add all necessary data.
Note that the foods are identified by the animal's
name and family, so we have to look up the primary key.
"""

import mysql.connector
from database import login_info

db = mysql.connector.Connect(**login_info)
cursor = db.cursor()

cursor.execute("""DROP TABLE IF EXISTS food""")
cursor.execute("""
    CREATE TABLE food (
        id INTEGER PRIMARY KEY AUTO_INCREMENT,
        anid INTEGER,
        feed VARCHAR(20),
        FOREIGN KEY (anid) REFERENCES animal(id))
    """)

data = [('Ellie', 'Elephant', ['hay', 'peanuts']),
        ('Gerald', 'Gnu', ['leaves', 'shoots']),
        ('Gerald', 'Giraffe', ['hay', 'grass']),
        ('Leonard', 'Leopard', ['meat']),
        ('Sam', 'Snake', ['mice', 'meat']),
        ('Steve', 'Snake', ['mice', 'meat']),
        ('Zorro', 'Zebra', ['grass', 'leaves'])]

for name, family, foods in data:
    cursor.execute("SELECT id FROM animal WHERE name=%s and family=%s",
                   (name, family))
    id = cursor.fetchone()[0]
    for food in foods:
        cursor.execute("""INSERT INTO food (anid, feed)
                        VALUES (%s, %s)""", (id, food))
    db.commit()
    print("Processed", name, family, id)
```

Note Unlike **tablepop.py**, this program does not require that the table be created before it is run. To remove any uncertainty about the state of the table, the **DROP TABLE IF EXISTS** statement is present to make sure that no food table exists when the **CREATE TABLE** statement is executed. If no such table exists then the **DROP TABLE IF EXISTS** statement has no effect.

When you run the program, it prints out each animal's details:

OBSERVE: Output from addfood.py

```
Processed Ellie Elephant 1
Processed Gerald Gnu 2
Processed Gerald Giraffe 3
Processed Leonard Leopard 4
Processed Sam Snake 5
Processed Steve Snake 6
Processed Zorro Zebra 7
```

So now you have a record of which animals eat which foods. Again, this is expressed as a relationship: animals eat food (one-to-many), food is-eaten-by animal (many-to-one). Now try a few queries through the interactive window. Use mysql to query the database:

INTERACTIVE SESSION:

```
mysql> SELECT * FROM food;
+----+------+---------+
| id | anid | feed    |
+----+------+---------+
|  1 |    1 | hay     |
|  2 |    1 | peanuts |
|  3 |    2 | leaves  |
|  4 |    2 | shoots  |
|  5 |    3 | hay     |
|  6 |    3 | grass   |
|  7 |    4 | meat    |
|  8 |    5 | mice    |
|  9 |    5 | meat    |
| 10 |    6 | mice    |
| 11 |    6 | meat    |
| 12 |    7 | grass   |
| 13 |    7 | leaves  |
+----+------+---------+
13 rows in set (0.04 sec)

mysql> SELECT name, family, feed
    -> FROM animal JOIN food ON animal.id=food.anid
    -> WHERE feed IN ('meat', 'leaves');
+---------+---------+--------+
| name    | family  | feed   |
+---------+---------+--------+
| Gerald  | Gnu     | leaves |
| Leonard | Leopard | meat   |
| Sam     | Snake   | meat   |
| Steve   | Snake   | meat   |
| Zorro   | Zebra   | leaves |
+---------+---------+--------+
5 rows in set (0.05 sec)

mysql> SELECT feed
    -> FROM animal JOIN food ON animal.id=food.anid
    -> WHERE name='Sam' AND Family='Snake';
+------+
| feed |
+------+
| mice |
| meat |
+------+
2 rows in set (0.04 sec)

mysql> SELECT COUNT(*) FROM food WHERE feed='meat';
+----------+
| COUNT(*) |
+----------+
|        3 |
+----------+
1 row in set (0.04 sec)

mysql>
```

The output shows that each row in the food table contains the relevant animal id. By joining the animal table to the food table on equality of animal id, we produce output containing one row for each combination of animal and food. A given animal can be associated with multiple foods, and the animal data is duplicated as many times as necessary, once for each related food row. That's how the SQL JOIN feature works.

Using Relational Data in Python

For almost thirty years now, the relational database has been the dominant model for storing persistent data. As you saw in an earlier lesson, Python has its own mechanisms, which are great when only Python programs are concerned, but not so useful when multiple languages must be used. In addition, there is a huge amount of "legacy data" already stored in databases, and Python would not be a very good programming language if it couldn't make use of that data. Of course now you understand that it can, although you've only used MySQL, there are Python drivers for almost every imaginable database.

When you use a database (or any other persistent data store) you are effectively creating a model of selected portions of the world. Though the model describes only those aspects of the world that are of interest to your application. For instance, when writing a hospital information system, you would probably want to know the names and birth dates of the patients, but most likely not their favorite football team or the kind of car they drive. Similarly, you would probably want to know how many beds are in each ward, but the color the walls are painted would not be particularly relevant.

The value of these models is that if you can keep them up to date (by changing them as the world changes—reducing the stock quantity of a product when some is sold, for example—you can *answer questions about the real world by querying the model*. If someone sends an order in for six widgets and you only have three in stock, you can respond by telling the customer there will be a slight delay, and then order more widgets from your supplier without having to walk to the warehouse and check the physical stock.

Metadata: Data about Data

One of the remarkable things about relational structures is that they're powerful enough to describe other relational structures. Since a database is packed full of routines to handle relational structures, it makes sense that most relational database management systems (RDBMSs) actually store a relational description of the application data structures they are used to create.

Note This description is often called the *data dictionary*, or the *system catalog*, or various other names. Many databases even allow you to retrieve data from the data dictionary, thereby allowing you to query the structure of the database (the data dictionary is comprised of tables, after all). If you are given access to the data dictionary, please remember *never* to try to update those tables directly yourself (unless you happen to be an experienced database administrator) —that's the RDBMS's job!

If you want to experiment with databases on your own computer, take a look at a standard library module, **sqlite3**, that lets you create and use relational structures without the complexity of an external database server and client/server communications. All data are stored in files held on the same computer that the programs run on. **sqlite3** has some limitations and a few quirks but it's a good place to start, and has been used to support many production programs.

Phew! Let's take a little break before moving on to the next lesson... okay, break's over. Let's go!

Database Hints and Tricks

Representing Data Rows

The last lesson focused on getting data into and out of a database. Now we'll go over some different techniques that make it more convenient to use our data, by treating relational data just like other data in our programs. We already learned that after creating a cursor from a database connection, we pass SQL to the cursor's **execute()** method. If the SQL statements produce data, we call an additional cursor method to retrieve the data. Most database cursors have three methods to retrieve data from the query results. Each data row is a tuple containing an element for each column in the query's result.

Method Name	Functionality
fetchone()	Returns the next database row from the result set. If no rows are left, it returns **None**.
fetchmany(*n*)	Returns a list of up to *n* rows. If the result set is exhausted, it returns an empty list.
fetchall()	Returns a list of all rows remaining in the result set.

Working With Tuples

While you can deal with the data as tuples, it's not always the most convenient technique. The issue with tuples is that you need to use a numeric index to retrieve elements. This can make your code unreadable, as this first coding exercise will show. Create a **DatabaseHints** project and assign it to your **Python2_Lessons** working set. Then, copy your **database.py** from **HandlingDatabases/src** to **DatabaseHints/src** to make your login information available to programs in this lesson's folder. Create **datatest.py** in the **DatabaseHints/src** folder as shown:

CODE TO TYPE:

```
"""
Demonstration of indexed access to data elements.
"""
import mysql.connector
from database import login_info

db = mysql.connector.Connect(**login_info)
cursor = db.cursor()

fmt = "{0:10} {1:10} {2:6}"
print(fmt.format("Animal", "Weight", "Family"))
print("-"*28)
cursor.execute("SELECT * FROM animal")
for animal in cursor.fetchall():
    print(fmt.format(animal[1], animal[3], animal[2]))
```

This code produces a listing of the animals' names, weights, and families as you might expect:

OBSERVE: Output from datatest.py

```
Animal     Weight Family
----------------------------
Ellie        2350 Elephant
Gerald       1400 Gnu
Gerald        940 Giraffe
Leonard       280 Leopard
Sam            24 Snake
Steve          35 Snake
Zorro         340 Zebra
```

The code uses a cursor's **execute()** method to request all animal data, and then iterates over the list of tuples returned by the cursor's **fetchall()** method. But looking at the last line of the code, it isn't at all obvious that **animal[1]** is the animal's name, **animal[3]** is its weight, and **animal[2]** represents the animal family. Since you know that readability is one of the most important aspects of code, it would be good to allow access to data elements by name. We can do that using various Python features.

One way to do it that will immediately improve the readability of our code, is to use an unpacking assignment in the **for** loop that iterates over the result set. At the same time, we'll change the SQL to explicitly retrieve only the fields we want. Each element of the (three-element) tuple is stored in its own variable, thanks to the unpacking assignment. This makes the code a bit easier to read, but it does not effect its result at all. This updated version of the code should produce exactly the same output. In **datatest.py**, type the code below as shown:

CODE TO TYPE:

```
"""
Demonstration of indexed access to data elements.
"""
import mysql.connector
from database import login_info

db = mysql.connector.Connect(**login_info)
cursor = db.cursor()

fmt = "{0:10} {1:6} {2:10}"
print(fmt.format("Animal", "Weight", "Family"))
print("-"*28)
~~cursor.execute("SELECT * FROM animal")~~
~~for animal in cursor.fetchall():~~
~~    print(fmt.format(animal[1], animal[3], animal[2]))~~
cursor.execute("SELECT name, weight, family FROM animal")
for name, weight, family in cursor.fetchall():
    print(fmt.format(name, weight, family))
```

Save and run it. You'll see the same results.

Representing Tables as Classes

Another way to make the code more comprehensible is to create an object for each row that has attributes with the same names as the columns, to hold the data elements retrieved from the database. Then we'll begin to see that the rows returned from a query are actually data objects. In the **DatabaseHints/src** folder, create **animal.py** as shown:

CODE TO TYPE:

```
"""
animal.py: a class to represent an animal in the database
"""
class Animal:

    def __init__(self, id, name, family, weight):
        self.id = id
        self.name = name
        self.family = family
        self.weight = weight
```

This class has no tests. We need to write some, quickly! Instead of getting into all the formality of unit tests, we can include a basic self-test. This will allow us to tailor the way an Animal appears when printed, by providing a **__repr__()** method to meet our own specifications. Modify **animal.py** as shown:

CODE TO TYPE:

```
"""
animal.py: a class to represent an animal in the database
"""
class Animal:

    def __init__(self, id, name, family, weight):
        self.id = id
        self.name = name
        self.family = family
        self.weight = weight

    def __repr__(self):
        return "Animal({0}, '{1}', '{2}', {3})".format(
                self.id, self.name, self.family, self.weight)

if __name__ == "__main__":
    import mysql.connector
    from database import login_info
    db = mysql.connector.Connect(**login_info)
    cursor = db.cursor()
    cursor.execute("SELECT id, name, family, weight FROM animal")
    animals = [Animal(*row) for row in cursor.fetchall()]
    from pprint import pprint
    pprint(animals)
```

Save and run it. You'll see this:

OBSERVE: Output from the animal.py tests

```
[Animal(1, 'Ellie', 'Elephant', 2350),
 Animal(2, 'Gerald', 'Gnu', 1400),
 Animal(3, 'Gerald', 'Giraffe', 940),
 Animal(4, 'Leonard', 'Leopard', 280),
 Animal(5, 'Sam', 'Snake', 24),
 Animal(6, 'Steve', 'Snake', 35),
 Animal(7, 'Zorro', 'Zebra', 340)]
```

Take a closer look:

OBSERVE: animal.py

```
"""
animal.py: a class to represent an animal in the database
"""
class Animal:

    def __init__(self, id, name, family, weight):
        self.id = id
        self.name = name
        self.family = family
        self.weight = weight

    def __repr__(self):
        return "Animal({0}, '{1}', '{2}', {3})".format(
                self.id, self.name, self.family, self.weight)

if __name__ == "__main__":
    import mysql.connector
    from database import login_info
    db = mysql.connector.Connect(**login_info)
    cursor = db.cursor()
    cursor.execute("SELECT id, name, family, weight FROM animal")
    animals = [Animal(*row) for row in cursor.fetchall()]
    from pprint import pprint
    pprint(animals)
```

The program now defines the representation of an **Animal** by implementing a **__repr__()** method for Animal. The **animals** list is created in a list comprehension that provides individual arguments to the Animal creation using Python's "*" feature. As we learned earlier, Python's "*" feature takes a tuple or list and turns it into a series of individual arguments, as required by the Animal class's **__init__()** method. The **pprint()** function, imported from the **pprint** module, displays the representation of each list element by calling the **__repr__()** method.

This test isn't perfect, but it does cover most basic functionality. A silently-passing test is usually better. (Can you think of a way to silence the testing? Consider the **exec()** function). The test code also shows you one way to create an instance of this class from a row in a database table. Keep in mind that this method depends on the precise order of the fields in the database table, which isn't always a given. Someone might change the structure of the database without your knowledge, which could cause problems.

You can go further by defining a function that returns a tailored class, of which you can create instances to represent each row. To create the class, you would call:

RC = RecordClass("animal", "id name family weight")

Once you create the class, you would create instances of that class by calling the class with values for each of the named columns as follows:

for row in cursor.fetchall(): row_record = RC(*row)

You have the power to go in many different directions with Python objects. As a relative newcomer to the programming scene, you might sometimes find yourself almost paralyzed by the limitless number of options you have. Don't panic. In almost every case, the best way to deal with the quandary is to go ahead and write something. If it needs to be changed later, that's fine—your tests should save you from big mistakes.

Going further, you'll consider the best way to create the Animal objects, and which methods they should have. And what kind of objects should those methods to return? It seems like it would be a good idea to have **read()** return an Animal instance, but is it appropriate for a method of Animal to return an animal? And what arguments should **read()** be capable of accepting?

Should **readAnimal()** be a function instead of an Animal method? How about **write()**? Or should that be **save()**? Is there any really important value for that method to return? Maybe the names of the written fields?

Of course, you don't always want every column of every row. Suppose you only wanted to retrieve certain columns; would it help to keep the names of those columns somewhere, and build the column names into the query somehow? That way you could have queries that didn't bring unnecessary data into memory, for example. This is not only possible, it's what we're going to do next!

Manipulating SQL in Python

Python objects have a defined life-cycle which generally begins by calling the type's **__new__()** method, then calling the **__init__()** method of the "instance" returned by that. But most objects' behavior is determined by the methods you write. You can write their **__new__()** and **__init__()** methods if you like. Once you know what you're doing, you can pretty much install your own logic and have objects behave according to your plan.

So, how would you like a query to behave? Do you want your query to be on just a single table? If not, you will need to generate *JOINs*—if there are *n* tables, there must be *n*-1 JOIN conditions. Do you want to be able to determine which columns from which of the joined tables should be read in, and updated when written? Do you want to be able to read and write those objects, at least by primary key?

There are some generic solutions to these problems, but those frameworks can be intimidating at first. With your knowledge of Python, you already understand some techniques that make the database data easier to handle. It's good to have a range of techniques at your command for different situations. In order to implement those techniques, you'll need to understand how Python can be used to create SQL statements. Check it out:

OBSERVE: Generic SQL SELECT Query

```
SELECT column, ... FROM relation WHERE conditions
```

The **relation** being queried is often a **table**, though you can query a **join** as well. The **column** list in the statement may consist of simple column names or qualified names. If two or more of the tables in a query possess columns with the same name, these columns can only be referred to using *tablename.columnname* syntax. You don't necessarily need all columns of a table each time you reference a row, so you can make a case for having several different object types for a given table, each using a different set of columns.

Note Using only a subset of the columns of a table can be taken to its logical extreme by actually splitting the columns across multiple tables, of "commonly used" and "less commonly used" columns. The technical name for this is *vertical partitioning*. What do you imagine a *horizontal partitioning* might do? (Answer at bottom)

Suppose **cols** is the list of column names you want, **table** is the name of the table, and there are no other conditions on the data. The SQL statement you'd need to start with is:

"SELECT {0} FROM {1}".format(", ".join(**cols**), **table**)

The rows returned by this query have len(**cols**) elements, and the name of column *n* is **cols**[*n*].

In this next example, you'll generate the SQL from its component parts, and have a chance to observe how queries can be built. Type the code below into an interactive interpreter session, as shown:

INTERACTIVE SESSION:

```
>>> cols ="id name family".split()
>>> ", ".join(cols)
'id, name, family'
>>> table = "animal"
>>> "SELECT {0} FROM {1}".format(", ".join(cols), table)
'SELECT id, name, family FROM animal'
>>> condition1 = "id=7"
>>> conditions = [condition1]
>>> " AND ".join(conditions)
'id=7'
>>> conditions.append("family IS NOT NULL")
>>> " AND ".join(conditions)
'id=7 AND family IS NOT NULL'
>>> "SELECT {0} FROM {1} WHERE {2}".format(
... ", ".join(cols), table, " AND ".join(conditions))
'SELECT id, name, family FROM animal WHERE id=7 AND family IS NOT NULL'
>>>
```

Let's take a closer look at that last statement:

OBSERVE: SELECT Statement Building

```
"SELECT {0} FROM {1} WHERE {2}".format(", ".join(cols), table, " AND ".join(cond
itions))
```

The result of this expression is:

'SELECT id, name, family FROM animal WHERE id=7 AND family IS NOT NULL'

When a query joins multiple tables, there is always a chance that a name conflict will occur—the same column name might be defined in multiple tables. If you have enough information about the database, you can predict and avoid such conflicts by using the fully-qualified name *table.column*. The SQL interpreter will tell you when you make mistakes like this.

Let's say you have the column names and corresponding data items in lists. You can create a Python object for each of the rows retrieved with attributes of the same names as the columns (the column names must be named in acceptable Python style for the scheme to work properly). Earlier, we looked at how attribute assignment works on Python objects. Don't worry if your memory is a bit fuzzy on this. Just focus on this part: if *x* is some Python object, then the assignment ***x*.name = value** is pretty much equivalent to ***x*.__dict__['name'] = value**, which can also be expressed as **setattr(*x*, 'name', value)**.

Now, suppose the column names are "id," "name," and "email," and that you have a (three-element) data row holding a value for each attribute. There are various ways to modify a Python object. The object must be an instance of some user-defined class though, because built-in classes like **int** and **list** use a different mechanism to look up attributes. Type the code below as shown into an interactive interpreter console:

INTERACTIVE SESSION:

```
>>> COLS = "id name email".split()
>>> data = (1, "Steve Holden", "steve@holdenweb.com")
>>> class row:
...     pass
...
>>> r1 = row()
>>> for col, d in zip(COLS, data):
...   setattr(r1, col, d)
...
>>> dir(r1)
['__doc__', '__module__', 'email', 'id', 'name']
>>> r1.id, r1.name, r1.email
(1, 'Steve Holden', 'steve@holdenweb.com')
>>>
```

So now you know how to inject arbitrary attributes into a Python object. Writing three lines of code to create the object you want is pretty economical. But when the **__dict__** attribute is actually a standard Python dict, it has an **update()** method, which you can call with either a dict or a sequence of (key, value) pairs as its sole argument. The arguments are added to the original dict, overwriting the values of existing keys and adding new ones as necessary. This means you can achieve the same result even more efficiently. Continue your previous interactive session, typing the code below as shown:

INTERACTIVE SESSION:

```
>>> zip(COLS, data)
<zip object at 0x0116F738>
>>> dict(zip(COLS, data))
{'email': 'steve@holdenweb.com', 'id': 1, 'name': 'Steve Holden'}
>>> r2 = row()
>>> r2.__dict__.update(dict(zip(COLS, data)))
>>> r2.email
'steve@holdenweb.com'
>>> dir(r2)
['__doc__', '__module__', 'email', 'id', 'name']
>>> r3 = row()
>>> r3.__dict__.update(zip(COLS, data))
>>> dir(r3)
['__doc__', '__module__', 'email', 'id', 'name']>>>
```

As the **r3** example above demonstrates, the **dict.update()** method also accepts a sequence of (name, value) tuples as an argument, avoiding the unnecessary creation of a dict. This type of manipulation is common in some applications.

Armed with this knowledge, you can now write a class with a constructor call that takes the column names and data items as arguments, and returns an object with the attributes set. Keep in mind that database column names do not always follow exactly the same rules as Python names, so you might find tables that don't adapt well to this technique. There are often remedies you can apply at the database level to compensate for poor naming choices, but that topic is beyond the scope of this course.

A Data Row Class

Create **datarow.py** in your **DatabaseHints/src** folder as shown below:

CODE TO TYPE:

```
"""
datarow.py : implements a simple database record class
"""

class row:
    def __init__(self, cols, data):
        self.__dict__.update(zip(cols, data))
    def __repr__(self):
        return "user_record(id={0.id} name={0.name} email={0.email})".format(sel
f)

if __name__ == "__main__": # Simple self-test
    r1 = row(['id', 'name', 'email'],
             (1, "Steve Holden", "steve@holdenweb.com"))
    if r1.id != 1 or r1.name != "Steve Holden" or r1.email != "steve@holdenweb.c
om":
        print("TEST FAILED: id={0.id} name={0.name} email={0.email}".format(r1))
```

The test code demonstrates a feature of the string **.format()** method. You can see that it is not difficult to access the named attributes of the format arguments (which are themselves addressed by number). So, rather than passing three arguments to **format()**, you just pass one, and select the fields inside the format. If we had used this ability in the **animal.py** example earlier, we could have replaced this:

OBSERVE: Original __repr__() method for Animal class

```
def __repr__(self):
    return "Animal({0}, '{1}', '{2}', {3})".format(
            self.id, self.name, self.family, self.weight)
```

with the slightly more readable:

OBSERVE: Revised __repr__()method for Animal class

```
def __repr__(self):
    return "Animal({0.id!r}, {0.name!r}, {0.family!r}, {0.weight!r})".format(sel
f)
```

The **!r** at the end of each format specification tells the interpreter to substitute the object's **repr()** representation. (That's why strings will still be displayed with quotation marks around them, even though none appear in the format).

A More General-Purpose Approach

The row class developed in the preceding section works well enough, but the column names have to be passed in every time you create a new object. It would be more convenient to create a class with the column names already incorporated. You can do this by constructing the class inside a function, which takes the column and the table names as arguments. The function then returns the class after inserting the table name and the column names as class attributes. The function effectively becomes a "class factory," returning a slightly different class each time it is called.

We'll write some basic tests for the function we're going to create—this will allow us to to verify its operation. In the **DatabaseHints/src** folder, create **testClassFactory.py** as shown:

CODE TO TYPE:

```
import unittest
from classFactory import build_row

class DBTest(unittest.TestCase):

    def setUp(self):
        C = build_row("user", "id name email")
        self.c = C([1, "Steve Holden", "steve@holdenweb.com"])

    def test_attributes(self):
        self.assertEqual(self.c.id, 1)
        self.assertEqual(self.c.name, "Steve Holden")
        self.assertEqual(self.c.email, "steve@holdenweb.com")

    def test_repr(self):
        self.assertEqual(repr(self.c),
                         "user_record(1, 'Steve Holden', 'steve@holdenweb.com')"
)

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

Now, in the **DatabaseHints/src** folder, create **classFactory.py** as shown:

CODE TO TYPE:

```
"""
classFactory: function to return tailored classes
"""

def build_row(table, cols):
    """Build a class that creates instances of specific rows"""
    class DataRow:
        """Generic data row class, specialized by surrounding function"""
        def __init__(self, data):
            """Uses data and column names to inject attributes"""
            assert len(data)==len(self.cols)
            for colname, dat in zip(self.cols, data):
                setattr(self, colname, dat)
        def __repr__(self):
            return "{0}_record({1})".format(self.table, ", ".join(["{0!r}".forma
t(getattr(self, c)) for c in self.cols]))
    DataRow.table = table
    DataRow.cols = cols.split()
    return DataRow
```

Running the test program, you'll see two passing tests:

OBSERVE:

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

OK
```

You're really soaking up this information! You now you have enough knowledge to be able to query a database. Good job! But we still haven't talked about updating databases yet. There are three particularly important SQL statements that we'll want to consider: INSERT, UPDATE and DELETE. In upcoming lessons, we investigate how those statements can be automated on a case-by-case basis. For now, you're ready to leave the world of databases behind and immerse yourself in an entirely different technology: e-mail. See you in the next lesson...

Note Answer to earlier question "What does a horizontal partitioning do?" It splits the table up into commonly-used and less-commonly-used sets of *rows*. Back to question

Handling Electronic Mail Messages

Handling Email

In this lesson we'll learn how to create and send email messages. We'll start by creating a plain text email, and send out that email with Python's **email** and **smtplib** modules. We'll look at the source of an email, and get a quick overview of RFC 2822, the request for comments that specifies an email's format.

Once you have an understanding of plain text emails, we'll move on to messages that have attachments and multiple parts—*MIME messages*. Again, we'll be dealing with Python's email module, which contains classes for handling messages that are composed of multiple parts and types. Just like with plain text emails, we'll experiment with creating and sending MIME messages. We'll see how MIME messages are composed, and how you can manipulate them when you have a MIME message to pick apart.

An Example of Email Written to a File

First, let's write a plain text email file. Create a new **HandlingEmail** project and assign it to the **Python2_Lessons** working set. In the **HandlingEmail/src** folder, create a file named **example-email.txt** as shown:

CODE TO TYPE:

```
From: anybody@work.com
To: anybody@home.com
Subject: Handling Emails With Python


This email was sent using Python's smtplib!
```

Replace the "From:" address, *anybody@work.com*, with the email address you have registered with O'Reilly (where you receive email from OST). Replace the "To:" address, *anybody@home.com*, with the same O'Reilly-registered address, or any other address you can access. Also, take note of the format of the headers and the empty line separating the headers from the body.

Representing an Email with Message Objects

Python's **email** module contains **Message**—a class with instances that represent email messages. (You'll learn more about the structure of an email later in this lesson when you get an overview of RFC 2822.) A Message object has headers and payloads. Headers and the body are the two main parts of an email. You can access the headers using dictionary-like syntax, or you can use the Message class's instance methods. The Message class handles the object representation of an email; it does not actually have the functionality to send emails (that functionality is in the smtplib module).

The email module also has **FeedParser** and **Parser** classes. These objects allow you to parse a stream of characters or a file as an email. However, since instantiating a parser and then calling a parse method is such a common sequence of operations for creating Message objects, there are convenience functions in the email module that bypass the use of these two classes. Instead, you can create a Message object from a flat file by using the email module's built-in **message_from_file()** function. There is also a similar **message_from_string()** function. The next example shows the creation and usage of a Message object. It incorporates the plain text email that you created earlier. Type the code below into an interactive Python console as shown:

INTERACTIVE SESSION:

```
>>> import email, datetime
>>> msg = email.message_from_file(open(r'v:/workspace/HandlingEmail/src/example-email.t
xt'))
>>> msg['From']
'anybody@work.com'
>>> msg['from']
'anybody@work.com'
>>> msg['To']
'anybody@home.com'
>>> msg['Date'] = datetime.datetime.now().strftime("%d %b %Y %H:%M:%S -0600")
'5 Aug 2010 10:00:00 -0700'
>>> msg['Subject']
'Handling Emails With Python'
>>> msg.get('From')
'anybody@work.com'
>>> msg.get('from')
'anybody@work.com'
>>> msg
<email.message.Message object at 0x00BF8970>
>>> print(msg.as_string())
From: anybody@work.com
To: anybody@home.com
Subject: Handling Emails With Python
Date: 5 Aug 2010 10:00:00 -0700

This email was sent using Python's smtplib!

>>> msg['X-Holden-Web'] = "Root beer for everyone!"
>>> print(msg.as_string())
From: anybody@work.com
To: anybody@home.com
Subject: Handling Emails With Python
Date: 5 Aug 2010 10:00:00 -0700
X-Holden-Web: Root beer for everyone!
>>> msg.get_payload()
"This email was sent using Python's smtplib!\n"

>>>
```

The **message_from_file()** function takes an opened file, and reads the file's contents to create a new **Message** object. You access its headers using the same kind of indexing that you use with dicts (you can also add headers by indexing the same way—you are even allowed to add proprietary headers, as long as their names begin with **"X-"**).

Header access is case-**in**sensitive. You can refer to the From header as either "From" or "from," using mapping style accessors or the **get()** method. There are multiple methods for poking and prodding the header and body information in a Message object—**get_payload()**, **as_string()**, and so on.

Sending Emails with smtplib

So, now that you have a representation of an email as a Python object, how do you actually send an email? In order to send an email, you'll need access to a mail server. Public email services like Yahoo, hotmail, or gmail, offer you access to their mail servers. If you've ever set up an email client, like Outlook, Thunderbird, or mail.app, to work with your web mail account, you should be familiar with configuring an outgoing mail server. You'll need to know the host name and port of the mail server you're going to use when you send emails with Python. The **smtplib** module's SMTP class represents a connection to a mail server. It allows you to *connect to and send mail from* that server.

Note If you know where to find your regular email settings, you can use those same SMTP server settings in the next few exercises. Otherwise, use the settings in the example below. If you do that, *all outgoing mail from your account will be redirected automatically to the email address you registered with O'Reilly.*

Type the code below into an interactive Python console as shown:

INTERACTIVE SESSION:

```
>>> import smtplib
>>> srv = smtplib.SMTP('mail.oreillyschool.com', 25)
>>> srv.sendmail(msg['From'], msg['To'], msg.as_string())
{}
>>> srv.quit()
(221, b'Service Closing transmission')
>>>
```

Note You may see a warning that you are attempting to send spam. This is a security feature of our SMTP server; we are currently working on a solution.

When you instantiated the SMTP object **srv**, you passed a host name and a port to its constructor. An alternative would be to instantiate the object and then immediately call its **connect()** method. If you are using a mail server that requires authentication, you'll need to call **login()** (with a username and a password as its arguments) before using the **sendmail()** method. Finally, as its name implies, **sendmail()** actually transmits your message. The From and To addresses must be supplied as the first two arguments, and the entire message as a string must be passed in as the third argument. We used the **as_string()** method to convert the entire message—the headers and the body—into a string. The entire message is required, including the headers; **get_payload()** would not be sufficient. Finally, you must call **quit()** to close your connection to your mail server.

You should receive the message that you just sent in the destination "To:" email account. Most email clients allow you to view an email's source. If you examine the source of the email that was sent, you'll get something like this:

OBSERVE:

```
Delivered-To: smtplib.example@gmail.com
Received: by 10.229.248.19 with SMTP id me19cs11861qcb;
    Thu, 4 Aug 2010 06:16:42 -0700 (PDT)
Received: by 10.227.69.17 with SMTP id x17mr4348340wbi.171.1273151801377;
    Thu, 5 Aug 2010 06:16:41 -0700 (PDT)
Return-Path: <smtplib.example@yahoo.com>
Received: from smtp112.plus.mail.re1.yahoo.com (smtp112.plus.mail.re1.yahoo.com [69.147
.102.75])
    by mx.google.com with SMTP id p18si2812439wbc.13.2010.05.06.06.16.39;
    Thu, 5 Aug 2010 06:16:40 -0700 (PDT)
Received-SPF: pass (google.com: best guess record for domain of smtplib.example@yahoo.c
om designates 69.147.102.75 as permitted sender) client-ip=69.147.102.75;
Authentication-Results: mx.google.com; spf=pass (google.com: best guess record for doma
in of smtplib.example@yahoo.com designates 69.147.102.75 as permitted sender) smtp.mail
=smtplib.example@yahoo.com; dkim=pass (test mode) header.i=@yahoo.com
Received: (qmail 71453 invoked from network); 6 May 2010 13:16:39 -0000
DomainKey-Signature: a=rsa-sha1; q=dns; c=nofws;
s=s1024; d=yahoo.com;
h=DKIM-Signature:Message-ID:Received:X-Yahoo-SMTP:X-YMail-OSG:X-Yahoo-Newman-Property:F
rom:To:Subject:Date;
b=2tutdYAS4lFp/y5bosZZbKefffTkEYgEzwkuBVBotA/MwnbX70g0+xWuNN2Fv9PqQNYkmL817pOEJJdWOqXmE
QUnp1FOkACuXG7B8UWbjzJmJLhbncuWd9tvXKPqtYc0PTXeGT+8Uy1t0fJGi38p3UYHxgH1vM5+VuDEQwT3W8Y=
  ;
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed; d=yahoo.com; s=s1024; t=127315179
9; bh=mBI+mFk/NBVawMtbV/D/wFxf8YugJRFLgkauQ63aW3I=; h=Message-ID:Received:X-Yahoo-SMTP:
X-YMail-OSG:X-Yahoo-Newman-Property:From:To:Subject:Date; b=p5sdatt7A9NABwx85pQE0yfN9vK
3BXUgAcFm7rN/v4zjCn2TxXKYvekLaGuNj3La8kl71pbf5Xv6vPjRKbcIizuNoXRnuB3lr6aR75rqzVZexRFHDM
jIKYnI9YyM5XemXbmG71WVAhEThkGm+K0TH4EhVvpNErLHo/y6cNtjQt0=
Message-ID: <317179.69250.qm@smtp112.plus.mail.re1.yahoo.com>
Received: from [192.168.1.146] (smtplib.example@XXX.XXX.XXX.XXX with plain)
    by smtp112.plus.mail.re1.yahoo.com with SMTP; 05 Aug 2010 06:16:39 -0700 PDT
X-Yahoo-SMTP: TvYTIr2swBBLfJ4hwbbruqy1ImdZ_uFJ9iC3Ww--
X-YMail-OSG: 1xdPB3cVM1mWl_7QIy3YY_1iLhS0cF29P0hOsaItTnh2cV5
AVGlSBuGUl30V8SuFKhKicU3FPPX5wCnZrzWz_I2anv4G3n.Mnak.bqWkyOj
Wa_T36GBd8PlXAIEMVRLnjBd3DaqEQCu3DgDP_5_w3u4CmwIrHI6pkDbGd3o
PT9xapGWr6M79XG2JE_SKC5VdCE8SvksSGfmtxX0mIZtwB61ZbhnlY5WOuLL
aHPML.XnABew_SwVbIGCARyGniU7.p_gz9DxmLnk3j64BCDa1ZGigG0w1bJ1
iyF.3uSWgsVG5OK03UGra6w_BjeSbDNaSGzM0jYG8KVFRR81DsotekR5O.3E
W99v26BEU
X-Yahoo-Newman-Property: ymail-3
From: anybody@work.com
To: anybody@home.com
Subject: Handling Emails With Python
Date: 5 Aug 2010 10:00:00 -0700

This email was sent using Python's smtplib!
```

Do you see the "X-Holden-Web" header in your message?

RFC 2822

The email's source looks a bit complicated at first glance. You normally don't see all of that stuff when you send and receive emails; mail clients like Outlook, Thunderbird, and gmail, remove all of the nitty gritty details. But in order to enable your program to send email, you'll need to be familiar with the specifications of an email's syntax. These specifications can be found in documents called RFCs—Requests for Comments. RFC 2822 contains the information you need to use Python's email-handling modules.

There's a lengthy, detailed standard for RFCs. For more information, refer to the Python library.

Essentially, RFC 2822 is the standard that specifies the message content format to be passed between email systems. A message is actually a series of characters. According to RFC 2822, a message has two parts: the headers and the body (which is optional). Think of the headers as an envelope and the body as the letter. The envelope, or headers, contain all of the information necessary for sending the message—the sender, the recipient, the date the message was created, and so on. The contents, or body (also referred to as the payload), is the actual message to be transmitted.

The header is separated from the payload by exactly one blank line (two consecutive line breaks). Line breaks can be represented by different characters, or in some cases, by combinations of characters. The RFC 2822 specification for emails uses the carriage return and line feed pair (CRLF) to represent a line break. Two consecutive CRLFs separate an email's headers from its body.

RFC 2822 is not the final word on email. Subsequent RFCs refine and clarify standards further. For example, RFCs 2045 through 2049 describe sending structured data, such as images and audio, via email. These RFCs, known together as Multipurpose Internet Mail Extensions (MIME), extend the definition of an email body. For now, RFC 2822 is enough to get us started with Python's email module. The four headers that we'll use are Orig-date, From, To, and Subject:

Field Name	Example
orig-date	Date: 24 Apr 2010 10:00:00 -0700
from	From: someone@domain.bar
to	To: foo@example.bar
subject	Subject: Hello

Take another look now, at the source of the email that you sent and received earlier. Pick out the fields that were required. Look at all of the headers that were inserted by the mail client and the mail server! Find the blank line that separates the headers from the body.

MIME Messages

So far, you've used string representations of Message objects to send emails with smtplib. But you could as easily have skipped parsing your email text into a Message object, and just passed a string directly from your file to the **sendmail()** method. The basic RFC 2822 format can make using a Message object to represent a plain text email seem like overkill.

The real value of the Message object abstraction will become more apparent when you use it to create emails that have multiple parts, contain non-English text (that is, character sets other than ASCII), or have non-text attachments. MIME is a set of standards that allows emails to contain those elements. Incorporating MIME requires some modification to the way you send plain text emails. You'll need boundaries for multipart messages, and extra headers that specify which content you're sending. The MIME RFCs specify several headers that are not present in RFC 2822, such as Content-Type and MIME-Version. Rather than going through each MIME-related RFC, we'll start with an example of how to send a basic MIME message with Python's email module.

MIME Messages in Python use the Message class. **MIMEBase**, a subclass of Message, encapsulates common MIME Message functionality. MIMEBase, in turn, serves as the parent of a family of classes that provide functionality for specific MIME types. Our next example shows how to create a MIME Message that's composed of two other messages—a plain text message and an html message. Let's get going already! Create a container message that holds the two text messages. Type this code into an interactive Python console:

INTERACTIVE SESSION:

```
>>> from email.mime.multipart import MIMEMultipart
>>> msg = MIMEMultipart()
>>> msg
<email.mime.multipart.MIMEMultipart object at 0x00BEB7D0>
>>> msg['To'] = 'anybody@home.com'
>>> msg['From'] = 'anybody@work.com'
>>> msg['Subject'] = 'Sending Multipart HTML Mail'
>>> print(msg.as_string())
Content-Type: multipart/mixed; boundary="===============1941993348=="
MIME-Version: 1.0
To: anybody@home.com
From: anybody@work.com
Subject: Sending Multipart HTML Mail

--===============1941993348==

--===============1941993348==--
>>> msg.get_content_type()
'multipart/mixed'
>>> msg.is_multipart()
True
>>> msg.get_boundary()
'===============2020970424=='
```

We used a specific MIME class rather than the Message class or the MIMEBase class to create our container. MIMEBase is an *abstract class*—it is not intended to be instantiated directly. Instead, we used a subclass of MIMEBase —in this case, MIMEMultipart. A MIMEMultipart object automatically sets a couple of headers for you: Content-Type and MIME-Version. You can see the values of these headers, along with the rest of the message, by calling the MIMEMultipart instance's **as_string()** method. Its headers indicate that its Content-Type is multipart/mixed. Alternatively, we can call **get_content_type()** to view the Content-Type value directly. We can also call the **is_multipart()** method to determine whether a message may be composed of subparts. Finally, the **get_boundary()** method shows the string that separates the different parts of a message.

Now that we have a container message, we can create the other two messages that we'll attach to it. Continue the interactive Python console session. Type the code below as shown:

INTERACTIVE SESSION:

```
>>> from email.mime.text import MIMEText
>>> text_msg = MIMEText('hello!', 'plain')
>>> html_msg = MIMEText('<strong>hello!</strong>', 'html')
>>> print(text_msg.as_string())
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

hello!
>>> print(html_msg.as_string())
Content-Type: text/html; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

<strong>hello!</strong>
>>> text_msg.is_multipart()
False
>>> html_msg.is_multipart()
False
>>> text_msg.get_content_type()
'text/plain'
>>> html_msg.get_content_type()
'text/html'
```

You created another two objects that are instances of a type-specific MIME class—MIMEText. The MIMEText constructor takes the payload as the first argument, and the subtype of the message as the second. Both messages are of type text, but one is text/plain, while the other is text/html. These objects do not have subparts; when you call **is_multipart()** on them, the result is False. By using **get_content_type()**, we see that the appropriate Content-Type headers are set. Again, the actual headers can be viewed by calling **as_string()** on either of these objects, or on the container message.

With these two messages created, we can insert them into the original multipart message that serves as the container. Continue the interactive Python console session. Type the code below as shown:

INTERACTIVE SESSION:

```
>>> msg.attach(html_msg)
>>> msg.attach(text_msg)
>>> msg.as_string()
'Content-Type: multipart/mixed; boundary="===============1941993348=="
MIME-Version: 1.0
To: anybody@home.com
From: anybody@work.com
Subject: Sending Multipart HTML Mail

--===============1941993348==
Content-Type: text/html; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

<strong>hello!</strong>
--===============1941993348==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

hello!
--===============1941993348==--'
>>> msg.get_payload()
[<email.mime.text.MIMEText object at 0x00C035B0>, <email.mime.text.MIMEText object at 0
x00C035D0>]
>>> messages = msg.get_payload()
>>> for m in messages:
...     print(m.get_content_type())
...
text/html
text/plain
>>> msg.walk()
<generator object walk at 0x00C048C8>
>>> for m in msg.walk():
...     print(m.is_multipart())
...     print(m.get_content_type())
...
True
multipart/mixed
False
text/html
False
text/plain
```

Using the **attach()** method, you can nest messages into your original container email. The submessages that you attached to your container email still retained their headers. Again, by using **as_string()**, you can see the headers of your message. This time though, you can see the headers of all of the messages because two them are subparts of the original. The boundary separates the messages.

When the **get_payload()** method is called on the top-level multipart message, the result is the list of the submessages it contains. The items in this list are also message objects. They are the text and html messages that you created. As is the case with regular Message objects, you can get the content type and the payload from them. In fact, everything you can do with Message objects, you can do with the submessages on this list. Going through nested messages by constantly calling **get_payload()** would be tedious. So for messages with complex nesting, the Message object supplies a **walk()** method which allows you to move through all of the messages parts and subparts.

Now that you have your multipart message constructed, you can send it using the smtplib module. Continue the interactive Python console session. Type the code below as shown:

INTERACTIVE SESSION:

```
>>> import smtplib
>>> srv = smtplib.SMTP('mail.oreillyschool.com', 25)
>>> srv.sendmail(msg['From'], msg['To'], msg.as_string())
{}
>>> srv.quit()
(221, b'Service Closing transmission')
>>>
```

When you check your email, the source will look something like this:

OBSERVE:

```
Delivered-To: smtplib.example@gmail.com
Received: by 10.229.184.72 with SMTP id cj8cs27998qcb;
    Wed, 5 Aug 2010 04:47:15 -0700 (PDT)
Received: by 10.229.230.76 with SMTP id jl12mr584775qcb.134.1273664835572;
    Wed, 5 Aug 2010 04:47:15 -0700 (PDT)
Return-Path: <smtplib.example@yahoo.com>
Received: from smtp107.plus.mail.re1.yahoo.com (smtp107.plus.mail.re1.yahoo.com [69.147
.102.70])
    by mx.google.com with SMTP id h8si95375qce.35.2010.05.12.04.47.12;
    Wed, 5 Aug 2010 04:47:14 -0700 (PDT)
Received-SPF: pass (google.com: best guess record for domain of smtplib.example@yahoo.c
om designates 69.147.102.70 as permitted sender) client-ip=69.147.102.70;
Authentication-Results: mx.google.com; spf=pass (google.com: best guess record for doma
in of smtplib.example@yahoo.com designates 69.147.102.70 as permitted sender) smtp.mail
=smtplib.example@yahoo.com; dkim=pass (test mode) header.i=@yahoo.com
Received: (qmail 14018 invoked from network); 5 Aug 2010 11:47:12 -0000
DomainKey-Signature: a=rsa-sha1; q=dns; c=nofws;
s=s1024; d=yahoo.com;
h=DKIM-Signature:Message-ID:Date:Received:X-Yahoo-SMTP:X-YMail-OSG:X-Yahoo-Newman-Prope
rty:Content-Type:MIME-Version:To:From:Subject;
b=lg6OxX1OKZAXksTKkzq8e1oO8ieAxFAappES61HNBM+0dbg+8W4EumPAipkzXc+FrfTxp9baEcuEOZHs6Nymh
CsSrGitG8YdH65q2DSyZ1nZfx+J8vTwnmBPWUERbDnb0jc0BjL8Yxp67CoPl5sQK70RQwRFA8zfuHVRKOF3CGY=
  ;
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed; d=yahoo.com; s=s1024; t=127366483
2; bh=9jXFcWGXd3kqMgAgceETpi+pfKBH4iw1lLP8TtNzJIo=; h=Message-ID:Date:Received:X-Yahoo-
SMTP:X-YMail-OSG:X-Yahoo-Newman-Property:Content-Type:MIME-Version:To:From:Subject; b=X
4f15CkB4Nncb53WVyv7W8DTvRH26barOUPgtgqRXSWaDoVp16Sh0auxPqto0wjHFYSb+k3RHQ7cDV2mmLLdacPc
bhm7nCnE1kcyjN+9YHyc1vDjcvSv4mC8cfCBh0BlBwPAQUpDzvEe5O8WHiAG+HHpeoRcrhdMk2vJ8zz75fc=
Message-ID: <666948.12858.qm@smtp107.plus.mail.re1.yahoo.com>
Date: Wed, 5 Aug 2010 04:47:12 -0700 (PDT)
Received: from [192.168.1.146] (smtplib.example@XXX.XXX.XXX.XXX with plain)
    by smtp107.plus.mail.re1.yahoo.com with SMTP; 5 Aug 2010 04:47:12 -0700 PDT
X-Yahoo-SMTP: TvYTIr2swBBLfJ4hwbbruqy1ImdZ_uFJ9iC3Ww--
X-YMail-OSG: .WEVb9sVM1lBMCoj.KTsuu4ud9OTVz2xFhg_0fgAIj82I6t
wG.lW3STKxIRYDBpPxsHAlHKn6nVLd_SHkOFi5Q3QqNDxvN1rURL3r4rLV5g
wal.7VIWZYVtB9dzHB3BTCUczn7WN_fojpSzk2muQn0DlpOLd_6_Pj2A1wgm
xGpHCqGgBSrvBzdtTAfWvSGqrkEzXpopsRBwrJcnODFF3W65LVua0x9b6Z41
zCr_HHODZIPOBdgOKOPDANhvpWoCY1hAHbsQoT4eLexZX63jSZ06VylQ1u_j
qlbqDaAFRgPltsNs4sxMASuTOjJr9dVK_vP5OtqQL11dxDqR6OJJrEQnNR96
cjM1EiGVD
X-Yahoo-Newman-Property: ymail-3
Content-Type: multipart/mixed; boundary="===============0044803118=="
MIME-Version: 1.0
To: smtplib.example@gmail.com
From: smtplib.example@yahoo.com
Subject: Sending Multipart HTML Mail

--===============0044803118==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

hello!
--===============0044803118==
Content-Type: text/html; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

<strong>hello!</strong>
--===============0044803118==--
```

There are a few headers in the source that you haven't encountered yet. Again, MIME adds several new headers to the email specification to describe the contents of an email. These new headers include:

- MIME-Version
- Content-Type
- Content-Transfer-Encoding
- Content-ID
- Content-Disposition

Only the first three—MIME-Version, Content-Type, and Content-Transfer-Encoding—are required for a MIME message.

MIME-Version indicates that the message conforms to the MIME standard. This is a signal to email clients and other email programs to perform the additional processing necessary to handle MIME messages. In practice, the value of this header is usually "1.0." It should appear at the top level of a message, though it can appear again if more MIME messages are attached to the original message (more on nesting messages later in this lesson).

OBSERVE: MIME-Version header

```
MIME-Version: 1.0
```

Once you have signaled that a message conforms to the MIME standards by using the MIME-Version header, you have to specify the type of content that is in the message. MIME messages are not limited to just text! The **Content-Type** header describes the kind of data that comprises the body. This description is made up of two parts, separated by a forward slash: type/subtype. The type is the general kind of data. The subtype is the format of that data. Some common Content-Type values are:

Type/Subtype; Parameter	Description
text/plain	A plain text message. In the absence of a Content-Type header, text/plain is usually assumed.
text/html	An HTML email—this tells your mail client that the email should be rendered as HTML, like a web page.
Message/RFC822	The Content-Type of another message; for example, in a reply, the original message may be attached.
Image/Jpeg	An image in jpeg format.
multipart/mixed; boundary=gc0y0pkb9ex	A message with multiple parts. The parts are separated by the boundary—gc0y0pkb9ex.

OBSERVE: Content-Type header

```
Content-Type: text/plain; charset="us-ascii"
```

Content-ID is a "world-unique" identifier for a part of MIME message. Just like the Message-ID header, this is usually automatically generated so that it is unique, regardless of when and where it was created. A message's Content-ID can be used in several different contexts. For example, it can aid in caching message parts, or it can serve as mechanism for maintaining references between different message parts.

OBSERVE: Content-ID header

```
Content-ID: <d41d8cd98f00b204e9800998ecf8427e@foo.bar>
```

The **Content-Disposition** header is an optional field that specifies how a MIME message part is displayed in your mail client. An **inline** part is automatically displayed in the regular flow of the message. An attachment part is not automatically displayed; instead, it requires some user action in order for it to be viewed (such as opening a pdf reader). The Content-Disposition header also allows you to specify a file name for an attachment. This is done by adding a filename parameter to the end of the header.

OBSERVE: Content-Disposition header

```
Content-Disposition: attachment; filename="files.zip"
```

Because binary data can't be transferred over some protocols, it has to be represented as ASCII text. For example, images and audio need a binary-to-text encoding in order to be sent. The **Content-Transfer-Encoding** header specifies which encoding—if any—was used. Base64 is a common binary-to-text encoding scheme. Right-click on the image below, select **Save Picture As..."** and save it as **v:/workspace/python-logo.png**.

OBSERVE: Content-Transfer-Encoding header

```
Content-Transfer-Encoding: base64
```

Go ahead and type the code below into an interactive Python console:

INTERACTIVE SESSION:

```
>>> import os
>>> from email.mime.image import MIMEImage
>>> fn = 'v:/workspace/python-logo.png'
>>> import mimetypes
>>> mimetypes.guess_type(fn)
('image/png', None)
>>> with open(fn, 'rb') as fp:
...     img = MIMEImage(fp.read())
...
>>> img['MIME-Version']
'1.0'
>>> img['Content-Type']
'image/png'
>>> img['Content-Transfer-Encoding']
'base64'
>>> img['Content-Disposition']
>>> img.get_filename()
>>> img.add_header('Content-Disposition', 'attachment', filename=os.path.basename(fn))
>>> img['Content-Disposition']
'attachment; filename="python-logo.png"'
>>> img.get_filename()
'python-logo.png'
```

There is a mimetypes module that contains a **guess_type()** method that guesses the content type of a file by looking at its extension. This can be handy for determining which MIME type class you should use to represent a message or file in your program, without doing content analysis. A few headers, such as MIME-Version, Content-Type, and Content-Transfer-Encoding, are automatically set by using the MIMEImage constructor. If you want to attach a file so that it's displayed as an attachment rather than inline, you can use the **add_header()** method and put in the appropriate header names and header values manually. The **add_header()** method takes as keyword arguments, the header name, the header value, and any optional parameters that you want to set for the header. Once you set the attachment's filename, you can retrieve the attachment's filename programmatically, using the **get_filename()** method.

In the Home Stretch

Using Python's email and smtplib modules, along with your knowledge about how emails are formatted, you can send emails as well as parse them. An email can be a single plain text email, or it can be a message that contains several sub-parts. Depending on what type of email you're sending, you'll set various headers that specify the details of your email—who it's from, who it goes to, what kind of content it contains, and so on. Python's email module offers a variety of classes, from the base Message class to the MIME* classes, to represent email messages. These classes offer conveniences like setting certain headers automatically, as well as methods that allow access to various parts of an email message (such as **get_payload()**, or **get()**), and methods that aid in the creation of messages (**add_header()**, **attach()**, etc.). Once you've created a Python representation of your message, you can use the smtplib module to connect to your mail server and send your message.

Wow, can you believe it? You've only got one more lesson to go. It seems like only yesterday you were learning about unittest...just look at you now! We've covered a lot of ground here, and you've done a great job. Still, if any part of it is confusing or you need some guidance, please call on your faithful instructor. See you in the next and final lesson!

Email Search and Display

A Really Useful Program

You have picked up lots of empowering skills in this course. You know how to build GUIs, you understand about various types of persistent storage, and you can handle email. In this last lesson, we'll build an email storage and retrieval mechanism, then attach that to a graphical user interface so that the user can enter various search criteria and click on a button to see a list of matching messages. Clicking on a message from the list will display the message body. Not only that, but the messages will be stored in a relational database on an entirely different computer from the one running the program. How cool is that?

So, how will we be able to search the email? Potentially by date and by partial match on the names and email addresses of senders. We'll also be able to extend this software to handle additional headers as retrieval keys.

We'll start by writing a program to create the necessary table in the MySQL database. We'll follow that with a library to handle storage and retrieval of information in the database, along with tests that exercise the storage and retrieval functionality. Finally, we'll write a GUI-based program to query the database and display the resulting messages.

A Basic Email Database

To store messages, we need a defined structure. A single table is the simplest store, so for now, we'll develop a single-table store. Once the basic message storage function is working (and tested!), we'll add columns to the table to enable new types of retrieval.

Message Identities

The modern email system is pretty good about allocating each individual message a globally-unique identification, which is carried in the **Message-Id** header. Here's a sample header from the author's current inbox:

OBSERVE: Sample Email Header

```
Message-ID: <20100529085040.32283.76682@betelnutz>
```

To keep relationships efficient and representations clear, each message in the database will have *two* unique columns. There will be **msgID** (an integer column automatically inserted as necessary by the RDBMS and used as the primary key) and **msgMessageID** (the globally-unique mail system identifier). **msgID** will be used to refer to a message wherever possible in the system. Messages themselves do sometimes refer to each other by the **Message-ID** value though, so that access path is worth putting into even a basic implementation.

This initial implementation records the **Message-ID** header value as a database column so you can use SQL to query it. Later, certain other information will also be extracted from the messages and recorded directly in the database for the same reason. This will allow full relational operations on that data, letting the database do the retrieval tasks for which it is optimized.

The store needs an *Application Programmer Interface* or *API*. This intimidating-sounding thing is actually a set of "how-to instructions for users of the message store." The fundamental operations of storage and retrieval are described in the API. Storing a message requires that the message be passed in to the storage function. Retrieving messages requires some kind of identity to be passed in, and for the function to return a message (or raise some sort of MessageNotFound exception). Since you have decided (well, okay, we've decided) to retrieve with both "msgID" and "MessageID," it makes sense to provide two functions. Here is the API that will help you accomplish your tasks:

Function	Purpose and interface
store(msg)	Adds the message to the store and returns its msgID value. If a message with the same value for the **Message-ID** header is already present in the store, the msgID of the existing message is returned.
message_by_id(id)	Returns the message whose primary key value is **id** or raises an exception if no such message is present.
message_by_messageid(message_id)	Returns the message whose MessageId value is **message_id** or

message_by_messageid(message_id)	raises an exception if no such message is present.

This is only an initial attempt to define the interface. Don't think of it as something set in stone. Often, after working with a newly-designed API for a little while, it turns out to be less than ideal. In that case, feel free to change it—programmer convenience is more important than strict adherence to existing APIs. Never be afraid to rework a portion of your design; well-designed systems are the result of experimentation and revision.

The Message Table

The message store is deliberately uncomplicated. You may be surprised at how sophisticated your queries have become by the end of the lesson. Initially there are just three columns: the **id** (automatically generated by the database), the **Message-ID** header value, and the **message** itself, represented in the most fundamental way: as the sequence of characters that was received over the network. This sequence can be parsed by the **email** module to produce **email.Message** objects. More columns will be added to the database table as its scope and capabilities grow.

Start MySQL at the terminal window, and create this table as shown:

INTERACTIVE SESSION:

```
mysql> CREATE TABLE message(
    ->     msgID INTEGER AUTO_INCREMENT PRIMARY KEY,
    ->     msgMessageID VARCHAR(128),
    ->     msgText LONGTEXT);
Query OK, 0 rows affected (0.18 sec)
```

That's it. You now have a table in which to store your messages. The message itself is stored as a character sequence in a LONGTEXT column. This particular type of column is designed to allow storage of arbitrary character strings. Email can be tricky stuff to store; not all messages will necessarily be in the same character set and no global default can be applied. (For example, sometimes the header data explains that certain portions of the message are in specific encodings).

Processing the message to create an **email.Message** object is the most efficient way to extract the **Message-Id** header. For now, if you want to find out anything else about the message once it's been stored, you'll need to read its text in from the database and parse it again. By ensuring all messages are parsed before entering the database, you guarantee they can be parsed upon retrieval.

At some point you may consider using some more efficient storage representation, such as a pickle. But an email is not necessarily best represented as a single object, so your initial approach will be the more conservative one outlined above. Remember—first, make it work! You can update the storage mechanism later if necessary.

You only need two pieces of data in order to insert a new row (representing a new message) into the message table: the Message ID and the bytestring representation of the message. The third column (primary key) will be populated automatically. So if you have the Message ID and the string representation in variables **message_id**, and **text** respectively, along with a database cursor in **curs**, the required statement would look like this:

OBSERVE: Inserting Data in the message Table

```
curs.execute("INSERT INTO message(msgMessageID, msgText) VALUES (%s, %s)",
             (message_id, text))
```

Beginnings of a Mail Database Module

Now that the table has been created, you need Python functionality that allows you to store, and retrieve, email messages.

Note Users of this API don't need to know how the data is stored in the database. If the API does not provide them with the features they need, knowledge of the structures allows the use of raw SQL, but it's better to try and avoid this. If only your code updates the database, then only you are responsible for its consistency. This is a good practice to adhere to in database management.

Before you insert the message into the database, you need to make sure that it isn't in there already. You could either retrieve the set of all rows having the given message ID and make sure that it is empty, or you could count all of the rows with a specific message-id. Since you need the primary key of the message when it *is* present in the database anyway (to return as the value of the function), you may as well try and retrieve it now.

Note When there is a possibility of retrieving lots of rows, but you only want to know how many there are, it's usually *much* more efficient to use the SQL **COUNT(*)** function we discussed earlier.

Create a **EmailSearch** project and assign it to the **Python2_Lessons** working set. Copy the **database.py** file from **python2_lesson10/src** to your **EmailSearch/src** folder. Then, in the **EmailSearch/src** folder, create a new Python file named **maildb.py** as shown:

CODE TO TYPE:

```
"""
Email message handling module: contains logic to store
email messages using a MySQL relational database.
"""
from database import login_info
import mysql.connector as msc
from email import message_from_string

conn = msc.Connect(**login_info)
curs = conn.cursor()

def store(msg):
    """
    Stores an email message, if necessary, returning its primary key.
    """
    message_id = msg['message-id']
    text = msg.as_string()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    result = curs.fetchone()
    if result:
        return result[0]
    curs.execute("INSERT INTO message (msgMessageID, msgText) VALUES (%s, %s)",
              (message_id, text))
    conn.commit()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    return curs.fetchone()[0]
```

This **maildb** module defines a single function, which takes a parsed **email.Message** and its textual equivalent as arguments. First, the function extracts the message's **Message-ID** header value, and attempts to retrieve the **msgid** of the message with that **message_id** (in case it already exists).

Sadly, we broke the first rule of test-driven development here! Remember it? "Only write code to make a failing test pass." But we wrote the code before writing the tests. Yes, we have led you down a dark and evil path. Don't let us do it again! Let's write the tests now. To get going in the right direction, we'll even add some tests that we know the existing code cannot pass, and then augment our code to make them pass in true test-driven development fashion.

In our consideration of email, we saw that the **Message-ID** header was a possible (candidate) primary key. While we do need to store this value as a column in the database (messaging systems often use **Message-ID** values to refer to other messages), it should not be the primary key—it's too long, and strings take longer to compare than numbers. So we made the primary key the **msgid** column, with values that are automatically allocated as rows are added to the database.

Now we can store messages in the database, right? Well, the only way to test storage is to retrieve data and verify that it agrees with what was stored. So we need a way of getting the information out—in fact we need two ways. We need to be able to retrieve a message with a given primary key, and with a given **Message-ID** header value. The code for each is somewhat similar.

Here's our testing strategy: each message is reconstituted from a text file, then stored using the

maildb.store() function. As the program iterates over the message files and stores the messages, it builds two dicts. The first one, **msgids**, maps **Message-ID** values to primary keys. The second, **message_ids**, maps primary key values to **Message-ID** values. The content of the dicts is used by the **test_msg_ids()** and **test_message_ids()** methods to verify that the expected message does indeed come back after retrieval by one or the other of the keys.

In the **EmailSearch/src** folder, create a **testMaildb.py** file as shown:

CODE TO TYPE:

```
"""
Read in and parse email messages to verify readability.

NOTE: This test creates the message table, dropping any
previous version and should leave it empty. DANGER: this
test will delete any existing message table.
"""

from glob import glob
from email import message_from_string
import mysql.connector as msc
from database import login_info
import maildb
import unittest

conn = msc.Connect(**login_info)
curs = conn.cursor()

TBLDEF = """\
CREATE TABLE message (
     msgID INTEGER AUTO_INCREMENT PRIMARY KEY,
     msgMessageID VARCHAR(128),
     msgText LONGTEXT
)"""
FILESPEC = "C:/PythonData/*.eml"

class testRealEmail_traffic(unittest.TestCase):
    def setUp(self):
        """
        Reads an arbitrary number of mail messages and
        stores them in a brand new messages table.

        DANGER: Any existing message table WILL be lost.
        """
        curs.execute("DROP TABLE IF EXISTS message")
        conn.commit()
        curs.execute(TBLDEF)
        conn.commit()
        files = glob(FILESPEC)
        self.msgids = {} # Keyed by message_id
        self.message_ids = {} # keyed by id
        for f in files:
            ff = open(f)
            text = ff.read()
            msg = message_from_string(text)
            id = self.msgids[msg['message-id']] = maildb.store(msg)
            self.message_ids[id] = msg['message-id']

    def test_not_empty(self):
        """
        Verify that the setUp method actually created some messages.
        If it finds no files there will be no messages in the table,
        the loop bodies in the other tests will never run, and potential
        errors will never be discovered.
        """
        curs.execute("SELECT COUNT(*) FROM message")
        messagect = curs.fetchone()[0]
        self.assertGreater(messagect, 0, "Database message table is empty")

    def test_message_ids(self):
        """
        Verify that items retrieved by id have the correct Message-ID.
        """
        for message_id in self.msgids.keys():
            pk, msg = maildb.msg_by_id(self.msgids[message_id])
            self.assertEqual(msg['message-id'], message_id)
    def test_ids(self):
        """
        Verify that items retrieved by message_id have the correct Message-ID.
        """
        for id in self.message_ids.keys():
            pk, msg = maildb.msg_by_message_id(self.message_ids[id])
            self.assertEqual(msg['message-id'], self.message_ids[id])

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

Save and run it. The tests fail, because the code calls two retrieval functions that we may not have written yet. In the code editor window, you'll see two error flags in the left margin:

Move your cursor over the red "X," and you'll see a tooltip that says something like "Undefined variable from import: msg_by_id". Even so, Eclipse will let you try and run the program. The resulting AttributeError exceptions cause the test to fail:

OBSERVE: Result of Initial Run of testMaildb.py

```
EE.
======================================================================
ERROR: test_ids (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 62, in test_ids
    pk, msg = maildb.msg_by_message_id(self.message_ids[id])
AttributeError: 'module' object has no attribute 'msg_by_message_id'

======================================================================
ERROR: test_message_ids (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 54, in test_message_id
s
    pk, msg = maildb.msg_by_id(self.msgids[message_id])
AttributeError: 'module' object has no attribute 'msg_by_id'

----------------------------------------------------------------------
Ran 3 tests in 0.688s

FAILED (errors=2)
```

The tests fail because there is no code present to implement the retrieval functions **msg_by_id()** and **msg_by_message_id()**. In this case, failure is great news—it means we're in proper test-driven development mode, now all we have to do is write those functions to pass the tests. Both of the retrieval functions return the message and its primary key. No matter how data is retrieved, store it using the primary key value to select the row to be updated.

The **msg_by_id()** function takes a primary key (id) value as its argument and executes a query to retrieve the message (along with the primary key). If this query returns an empty result set, the function raises a KeyError exception. Otherwise, **msg_by_id()** extracts the message text and its primary key from the database, and returns the primary key and a newly-parsed mail message.

This test mechanism is somewhat inefficient because it creates the table and then drops it for each individual test; it would be better to run the data creation once and then run each individual test. But we want to have two separate tests to make sure that a failure in one retrieval routine won't stop us from testing the other, so for now we'll put up with this bit of inefficiency.

Edit your **maildb.py** library to add this retrieval function as shown:

CODE TO TYPE:

```
"""
Email message handling module: contains logic to store and retrieve
email messages using a MySQL relational database.
"""
from database import login_info
import mysql.connector as msc
from email import message_from_string

conn = msc.Connect(**login_info)
curs = conn.cursor()

def store(msg):
    """
    Stores an email message, if necessary, returning its primary key.
    """
    message_id = msg['message-id']
    text = msg.as_string()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    result = curs.fetchone()
    if result:
        return result[0]
    curs.execute("INSERT INTO message (msgMessageID, msgText) VALUES (%s, %s)",
             (message_id, text))
    conn.commit()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    return curs.fetchone()[0]

def msg_by_id(id):
    """
    Return the (presumably singleton) message whose primary key is given
    or raise KeyError if no such message exists.
    """
    curs.execute("SELECT msgID, msgText FROM message WHERE msgID=%s", (id, ))
    result = curs.fetchone()
    if  not result:
        raise KeyError("Id {0} not found in store".format(id))
    id, text = result
    msg = message_from_string(text)
    return id, msg
```

With this new logic in place, one of the tests will succeed when you re-run **testMaildb.py**:

OBSERVE:

```
E..
======================================================================
ERROR: test_ids (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 62, in test_ids
    pk, msg = maildb.msg_by_message_id(self.message_ids[id])
AttributeError: 'module' object has no attribute 'msg_by_message_id'

----------------------------------------------------------------------
Ran 3 tests in 0.578s

FAILED (errors=1)
```

There is little difference between **msg_by_id()** and **msg_by_message_id()**. It is just a matter of using a slightly different condition on the query. Modify **maildb.py** by adding the code below as shown:

CODE TO TYPE:

```
"""
Email message handling module: contains logic to store and retrieve
email messages using a MySQL relational database.
"""
from database import login_info
import mysql.connector as msc
from email import message_from_string

conn = msc.Connect(**login_info)
curs = conn.cursor()

def store(msg):
    """
    Stores an email message, if necessary, returning its primary key.
    """
    message_id = msg['message-id']
    text = msg.as_string()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    result = curs.fetchone()
    if result:
        return result[0]
    curs.execute("INSERT INTO message (msgMessageID, msgText) VALUES (%s, %s)",
             (message_id, text))
    conn.commit()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    return curs.fetchone()[0]

def msg_by_id(id):
    """
    Return the (presumably singleton) message whose primary key is given
    or raise KeyError if no such message exists.
    """
    curs.execute("SELECT msgID, msgText FROM message WHERE msgID=%s", (id, ))
    result = curs.fetchone()
    if  not result:
        raise KeyError("Id {0} not found in store".format(id))
    id, text = result
    msg = message_from_string(text)
    return id, msg

def msg_by_message_id(message_id):
    """
    Return the (presumably singleton) message whose "Message-ID" is given
    or raise KeyError if no such message exists.
    """
    curs.execute("SELECT msgID, msgText FROM message WHERE msgMessageID=%s", (me
ssage_id, ))
    result = curs.fetchone()
    if  not result:
        raise KeyError("Message-Id {0} not found in store".format(message_id))
    id, text = result
    msg = message_from_string(text)
    return id, msg
```

Finally, all of our tests pass, and we can proceed to develop this basic library into something we can really use:

OBSERVE: Life is good when tests pass

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.580s

OK
```

Extending the Database's Retrieval Capabilities

According to our tests, we can now store email messages in a relational database and retrieve them either by primary key or Message-ID value. The Message-ID is extracted from the message when it is stored. It doesn't hurt to leave records lying around after a test, to allow testers to query the database manually and see what else can be done with the records, though the production installers prefer to have the tables left in a known empty state. It *certainly* doesn't hurt to know that the table passed its basic tests after installation. There are other pieces of information about the messages that you might like to store in the relational database to expand your retrieval capabilities even further. Specifically, you want to be able to retrieve messages sent between specific dates and/or times, and from specific senders, by name or address.

To accomplish that, we'll add a new column containing the message date. But it wouldn't be particularly useful to store it as a text column in the database, because the database cannot execute time-based calculations on strings. So instead, after you have extracted the **Date** header value from the parsed message, convert it into a Python **datetime.datetime** object, which the database driver will then convert into a MySQL DATETIME value, for storage in the database.

We'll modify the test program, adding a **msgDate** column to the table definition and add tests of the date retrieval function. We'll write that function later; it will look like this:

def msgs_by_date(mindate, maxdate)

Retrieval by date is different from retrieval by primary key or **Message-ID**—there is a real possibility that multiple messages will have the same date, causing the new function to return multiple records.

Our tests should work independently of the test data. To test the date routine, we'll change the date creation code in the **setUp()** method so that it also records the minimum and maximum datetime and a message count. Then we'll add a third test, **test_dates()**, that requests retrieval of all messages between the minimum and maximum datetimes, and verifies that the count is correct, and that all messages have the correct msgid values. Modify **testMaildb.py** as shown:

CODE TO TYPE:

```
"""
Read in and parse email messages to verify readability.

NOTE: This test creates the message table, dropping any
previous version and should leave it empty. DANGER: this
test will delete any existing message table.
"""
from glob import glob
from email import message_from_string
import mysql.connector as msc
from database import login_info
import maildb
import unittest
import datetime
from email.utils import parsedate_tz, mktime_tz

conn = msc.Connect(**login_info)
curs = conn.cursor()

TBLDEF = """\
CREATE TABLE message (
     msgID INTEGER AUTO_INCREMENT PRIMARY KEY,
     msgMessageID VARCHAR(128),
     msgDate DATETIME,
     msgText LONGTEXT
)"""
FILESPEC = "C:/PythonData/*.eml"

class testRealEmail_traffic(unittest.TestCase):
    def setUp(self):
        """
        Reads an arbitrary number of mail messages and
        stores them in a brand new messages table.

        DANGER: Any existing message table WILL be lost.
        """
        curs.execute("DROP TABLE IF EXISTS message")
        conn.commit()
        curs.execute(TBLDEF)
        conn.commit()
        files = glob(FILESPEC)
        self.msgids = {} # Keyed by message_id
        self.message_ids = {} # keyed by id
        self.msgdates = []
        self.rowcount = 0
        for f in files:
            ff = open(f)
            text = ff.read()
            msg = message_from_string(text)
            id = self.msgids[msg['message-id']] = maildb.store(msg)
            self.message_ids[id] = msg['message-id']
            date = msg['date']
            self.msgdates.append(datetime.datetime.fromtimestamp(mktime_tz(parse
date_tz(date))))
            self.rowcount += 1 # Assuming no duplicated Message-IDs

    def test_not_empty(self):
        """
        Verify that the setUp method actually created some messages.
        If it finds no files there will be no messages in the table,
        the loop bodies in the other tests will never run, and potential
        errors will never be discovered.
        """
        curs.execute("SELECT COUNT(*) FROM message")
        messagect = curs.fetchone()[0]
        self.assertGreater(messagect, 0, "Database message table is empty")
    def test_message_ids(self):
        """
        Verify that items retrieved by id have the correct Message-ID.
        """
        for message_id in self.msgids.keys():
            id, msg = maildb.msg_by_id(self.msgids[message_id])
            self.assertEqual(msg['message-id'], message_id)
            self.assertEqual(id, self.msgids[message_id])

    def test_ids(self):
        """
        Verify that items retrieved by message_id have the correct Message-ID.
        """
        for id in self.message_ids.keys():
            id1, msg = maildb.msg_by_message_id(self.message_ids[id])
            self.assertEqual(msg['message-id'], self.message_ids[id])
            self.assertEqual(id, id1)

    def test_dates(self):
        """
        Verify that retrieving records between the minimum and maximum dates
        returns an appropriate number of records.
        """
        mind = min(self.msgdates)
        mindate = datetime.date(mind.year, mind.month, mind.day)
        maxd = max(self.msgdates)
        maxdate = datetime.date(maxd.year, maxd.month, maxd.day)
        self.assertEqual(self.rowcount,
                         len(maildb.msgs_by_date(mindate=mindate,
                                                 maxdate=maxdate)))

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

Assigning this test the task of creating the table, ensures that the table definition stays up-to-date. This is actually the only way the tests can succeed—if SQL refers to a nonexistent column, the Python code that uses it will raise an exception.

Save and run it. The updated test fails, because we haven't updated the library yet. But hey, at least the original tests are still passing! The new test fails because it calls a function that we haven't written yet:

OBSERVE:

```
E...
======================================================================
ERROR: test_dates (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 83, in test_dates
    len(maildb.msgs_by_date(mindate=mindate,
AttributeError: 'module' object has no attribute 'msgs_by_date'

----------------------------------------------------------------------
Ran 4 tests in 0.672s

FAILED (errors=1)
```

We don't need to add much code to store the messages—the change looks bigger than it otherwise might because some operations have been re-ordered to avoid unnecessary work. We do need to import a couple of bits of code from **email.utils** and **datetime**. And, if the record isn't already present, the **store()** function extracts and converts the date, before storing it as an additional column in the table.

Now we need some way of retrieving the messages by date. We'll add a **msgs_by_date()** function that takes a minimum and/or a maximum date. The SQL that is generated makes sure that only one date will be provided. The parameters are dates rather than date-times, because we assume that humans are more interested in dates than times for most purposes. For the upper limit, we add a day to the given date and use a "less than" comparison. The code requires that at least one criterion be provided, and there is some logic to allow the code to work with either one or two conditions. Modify **maildb.py** as shown below:

CODE TO TYPE:

```
"""
Email message handling module: contains logic to store and retrieve
email messages using a MySQL relational database.
"""
from database import login_info
import mysql.connector as msc
from email import message_from_string
from email.utils import parsedate_tz, mktime_tz
from datetime import datetime, timedelta

conn = msc.Connect(**login_info)
curs = conn.cursor()

def store(msg):
    """
    Stores an email message, if necessary, returning its primary key.
    """
    message_id = msg['message-id']
    ~~text = msg.as_string()~~
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    result = curs.fetchone()
    if result:
        return result[0]
    date = msg['date']
    dt = datetime.fromtimestamp(mktime_tz(parsedate_tz(date)))
    text = msg.as_string()
    curs.execute("INSERT INTO message (msgMessageID, msgDate, msgText) VALUES (%
s, %s, %s)",
              (message_id, dt, text))
    conn.commit()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    return curs.fetchone()[0]

def msg_by_id(id):
    """
    Return the (presumably singleton) message whose primary key is given
    or raise KeyError if no such message exists.
    """
    curs.execute("SELECT msgID, msgText FROM message WHERE msgID=%s", (id, ))
    result = curs.fetchone()
    if  not result:
        raise KeyError("Id {0} not found in store".format(id))
    id, text = result
    msg = message_from_string(text)
    return id, msg

def msg_by_message_id(message_id):
    """
    Return the (presumably singleton) message whose "Message-ID" is given
    or raise KeyError if no such message exists.
    """
    curs.execute("SELECT msgID, msgText FROM message WHERE msgMessageID=%s", (me
ssage_id, ))
    result = curs.fetchone()
    if  not result:
        raise KeyError("Message-Id {0} not found in store".format(message_id))
    id, text = result
    msg = message_from_string(text)
    return id, msg

def msgs_by_date(mindate=None, maxdate=None):
    if not (mindate or maxdate):
        raise TypeError("Must provide at least one of mindate, maxdate")
    conds = []
    data = []
    if mindate:
        conds.append("msgDate >= %s")
        data.append(mindate)
    if maxdate:
        conds.append("msgdate < %s")
        data.append(maxdate+timedelta(days=1))
    sql = "SELECT msgid, msgText FROM message WHERE "
    sql += " AND ".join(conds)
    curs.execute(sql, tuple(data))
    result = []
    for id, text in curs.fetchall():
        result.append((id, message_from_string(text)))
    return result
```

Save and run it (from **testMaildb.py**) to verify that all of the tests now pass and also to confirm that we have implemented date-based storage correctly:

OBSERVE:

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.890s

OK
```

Tests all passed. Excellent. Proceed!

Practical Application

You might have thought you had the beginnings of a useful library with **maildb.py**, but the design is missing something—descriptions of the practical uses your program could fulfill using the library or **use cases**.

We know we can retrieve mail by date now, but typically we want to apply the date restrictions along with other constraints, like "sent by user@domain" or "recipients include user@domain." Before we go any further, we'll want to know more about the application that will be using the library.

You can always work directly with the database tables to provide a date-ordered listing of subjects. In the **EmailSearch/src** folder, create **mlist1.py** as shown:

CODE TO TYPE:

```
"""
Sample program to list subjects by date.
"""
from database import login_info
import mysql.connector
from email import message_from_string
conn = mysql.connector.Connect(**login_info)
curs = conn.cursor()
curs.execute("SELECT msgText FROM message ORDER BY msgDate")
for text, in curs.fetchall():
    msg = message_from_string(text)
    print(msg['date'], msg['subject'])
```

So, what are the retrieval requirements of this application? The intention is to allow the user to enter any or all of a start date, an end date, sender's name, and sender's email address, and then to list the dates and subject lines of each message. They should be able to click a message to display it.

As we saw earlier, the existing date field in the table allows us to select dates, but at present, we are not extracting the other necessary values—sender's name and email address—as database columns. We need to fix that. The sender's data are held in the **From** header. The format of the header data allows the inclusion of both a textual name and email address; the **email.utils** library has a **parseaddr()** function that we can use to move both pieces of information from the **From** header into a (name, address) tuple. That data can then be stored in two additional columns in the messages table. The code changes are subtle, particularly since we aren't adding any new retrieval routines this time around. Modify **maildb.py** as shown:

CODE TO TYPE:

```
"""
Email message handling module: contains logic to store and retrieve
email messages using a MySQL relational database.
"""
from database import login_info
import mysql.connector as msc
from email import message_from_string
from email.utils import parsedate_tz, mktime_tz, parseaddr
from datetime import datetime, timedelta

conn = msc.Connect(**login_info)
curs = conn.cursor()

def store(msg):
    """
    Stores an email message, if necessary, returning its primary key.
    """
    message_id = msg['message-id']
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    result = curs.fetchone()
    if result:
        return result[0]
    date = msg['date']
    name, email = parseaddr(msg['from'])
    dt = datetime.fromtimestamp(mktime_tz(parsedate_tz(date)))
    text = msg.as_string()
    curs.execute("""INSERT INTO message
                    (msgMessageID, msgDate, msgSenderName, msgSenderAddress, msg
Text)
                    VALUES (%s, %s, %s, %s, %s)""",
                    (message_id, dt, name, email, text))
    conn.commit()
    curs.execute("SELECT msgID FROM message WHERE msgMessageID=%s", (message_id,
 ))
    return curs.fetchone()[0]

def msg_by_id(id):
    """
    Return the (presumably singleton) message whose primary key is given
    or raise KeyError if no such message exists.
    """
    curs.execute("SELECT msgID, msgText FROM message WHERE msgID=%s", (id, ))
    result = curs.fetchone()
    if  not result:
        raise KeyError("Id {0} not found in store".format(id))
    id, text = result
    msg = message_from_string(text)
    return id, msg

def msg_by_message_id(message_id):
    """
    Return the (presumably singleton) message whose "Message-ID" is given
    or raise KeyError if no such message exists.
    """
    curs.execute("SELECT msgID, msgText FROM message WHERE msgMessageID=%s", (me
ssage_id, ))
    result = curs.fetchone()
    if  not result:
        raise KeyError("Message-Id {0} not found in store".format(message_id))
    id, text = result
    msg = message_from_string(text)
    return id, msg

~~def msgs_by_date(mindate=None, maxdate=None):~~
def msgs(mindate=None, maxdate=None, namesearch=None, addsearch=None):
    """
    Return a list of all messages sent on or after mindate and on or before maxd
ate.
    If mindate is not specified, there is no lower bound on the date, and simila
rly
    if maxdate is not specified, no upper bound. If namesearch is given, the
    result set is restricted to messages with sender names containing that strin
g. If
    addsearch is given, the result set is restricted to messages with email
    addresses containing that string.
    """
    ~~if not (mindate or maxdate):~~
~~        raise TypeError("Must provide at least one of mindate, maxdate")~~
    conds = []
    data = []
    if mindate:
        conds.append("msgDate >= %s")
        data.append(mindate)
    if maxdate:
        conds.append("msgdate < %s")
        data.append(maxdate+timedelta(days=1))
    if namesearch:
        conds.append("msgSenderName LIKE %s")
        data.append("%" + namesearch.strip().lower() + "%")
    if addsearch:
        conds.append("msgSenderAddress LIKE %s")
        data.append("%" + addsearch.strip().lower() + "%")
    ~~sql = "SELECT msgid, msgText FROM message WHERE "~~
    ~~sql += " AND ".join(conds)~~
    sql = "SELECT msgid, msgText FROM message"
    if conds:
        sql += " WHERE " + " AND ".join(conds)
    curs.execute(sql, tuple(data))
    result = []
    for id, text in curs.fetchall():
        result.append((id, message_from_string(text)))
    return result
```

Save it and run **testMaildb.py**. This revision breaks our existing tests. The library now references columns that have not been added to the database yet, so the driver complains during setup for each of the tests when we try to add a row. Also, pay attention to the change of function names in the module. Because the new retrieval function we wrote does more now that just retrieve mail by date, its name is something less specialized: **msgs**.

OBSERVE: test failures are induced by adding new columns in the library code

```
EEEE
======================================================================
ERROR: test_dates (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 51, in setUp
    id = self.msgids[msg['message-id']] = maildb.store(msg)
  File "V:\workspace\EmailSearch\src\maildb.py", line 30, in store
    (message_id, dt, name, email, text))
  File "C:\python\lib\site-packages\mysql\connector\cursor.py", line 307, in exe
cute
    res = self.db().protocol.cmd_query(stmt)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 137, in d
eco
    return func(*args, **kwargs)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 482, in c
md_query
    return self.handle_cmd_result(self._recv_packet())
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 175, in _
recv_packet
    MySQLProtocol.raise_error(buf)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 169, in r
aise_error
    raise errors.get_mysql_exception(errno,errmsg)
mysql.connector.errors.ProgrammingError: 1054: Unknown column 'msgSenderName' in
 'field list'

======================================================================
ERROR: test_ids (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 51, in setUp
    id = self.msgids[msg['message-id']] = maildb.store(msg)
  File "V:\workspace\EmailSearch\src\maildb.py", line 30, in store
    (message_id, dt, name, email, text))
  File "C:\python\lib\site-packages\mysql\connector\cursor.py", line 307, in exe
cute
    res = self.db().protocol.cmd_query(stmt)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 137, in d
eco
    return func(*args, **kwargs)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 482, in c
md_query
    return self.handle_cmd_result(self._recv_packet())
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 175, in _
recv_packet
    MySQLProtocol.raise_error(buf)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 169, in r
aise_error
    raise errors.get_mysql_exception(errno,errmsg)
mysql.connector.errors.ProgrammingError: 1054: Unknown column 'msgSenderName' in
 'field list'

======================================================================
ERROR: test_message_ids (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 51, in setUp
    id = self.msgids[msg['message-id']] = maildb.store(msg)
  File "V:\workspace\EmailSearch\src\maildb.py", line 30, in store
    (message_id, dt, name, email, text))
  File "C:\python\lib\site-packages\mysql\connector\cursor.py", line 307, in exe
cute
    res = self.db().protocol.cmd_query(stmt)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 137, in d
eco
```

```
    return func(*args, **kwargs)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 482, in c
md_query
    return self.handle_cmd_result(self._recv_packet())
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 175, in _
recv_packet
    MySQLProtocol.raise_error(buf)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 169, in r
aise_error
    raise errors.get_mysql_exception(errno,errmsg)
mysql.connector.errors.ProgrammingError: 1054: Unknown column 'msgSenderName' in
 'field list'

======================================================================
ERROR: test_not_empty (__main__.testRealEmail_traffic)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\EmailSearch\src\testMaildb.py", line 51, in setUp
    id = self.msgids[msg['message-id']] = maildb.store(msg)
  File "V:\workspace\EmailSearch\src\maildb.py", line 28, in store
    (message_id, dt, name, email, text))
  File "C:\python\lib\site-packages\mysql\connector\cursor.py", line 307, in exe
cute
    res = self.db().protocol.cmd_query(stmt)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 137, in d
eco
    return func(*args, **kwargs)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 482, in c
md_query
    return self.handle_cmd_result(self._recv_packet())
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 175, in _
recv_packet
    MySQLProtocol.raise_error(buf)
  File "C:\python\lib\site-packages\mysql\connector\protocol.py", line 169, in r
aise_error
    raise errors.get_mysql_exception(errno,errmsg)
mysql.connector.errors.ProgrammingError: 1054: Unknown column 'msgSenderName' in
 'field list'

----------------------------------------------------------------------
Ran 4 tests in 0.171s

FAILED (errors=3)
```

We need to update the test program and add the two more columns to the message table. Since we're familiar with adding columns now, let's bypass writing tests for these. Modify **testMaildb.py** as shown:

CODE TO TYPE:

```
"""
Read in and parse email messages to verify readability.

NOTE: This test creates the message table, dropping any
previous version and should leave it empty. DANGER: this
test will delete any existing message table.
"""
from glob import glob
from email import message_from_string
import mysql.connector as msc
from database import login_info
import maildb
import unittest
import datetime
from email.utils import parsedate_tz, mktime_tz

conn = msc.Connect(**login_info)
curs = conn.cursor()

TBLDEF = """\
CREATE TABLE message (
     msgID INTEGER AUTO_INCREMENT PRIMARY KEY,
     msgMessageID VARCHAR(128),
     msgDate DATETIME,
     msgSenderName VARCHAR(128),
     msgSenderAddress VARCHAR(128),
     msgText LONGTEXT
)"""
FILESPEC = "C:/PythonData/*.eml"

class testRealEmail_traffic(unittest.TestCase):
    def setUp(self):
        """
        Reads an arbitrary number of mail messages and
        stores them in a brand new messages table.

        DANGER: Any existing message table WILL be lost.
        """
        curs.execute("DROP TABLE IF EXISTS message")
        conn.commit()
        curs.execute(TBLDEF)
        conn.commit()
        files = glob(FILESPEC)
        self.msgids = {} # Keyed by message_id
        self.message_ids = {} # keyed by id
        self.msgdates = []
        self.rowcount = 0
        for f in files:
            ff = open(f)
            text = ff.read()
            msg = message_from_string(text)
            id = self.msgids[msg['message-id']] = maildb.store(msg)
            self.message_ids[id] = msg['message-id']
            date = msg['date']
            self.msgdates.append(datetime.datetime.fromtimestamp(mktime_tz(parse
date_tz(date))))
            self.rowcount += 1 # Assuming no duplicated Message-IDs

    def test_not_empty(self):
        """
        Verify that the setUp method actually created some messages.
        If it finds no files there will be no messages in the table,
        the loop bodies in the other tests will never run, and potential
        errors will never be discovered.
        """
        curs.execute("SELECT COUNT(*) FROM message")
        messagect = curs.fetchone()[0]
        self.assertGreater(messagect, 0, "Database message table is empty")

    def test_message_ids(self):
        """
        Verify that items retrieved by id have the correct Message-ID.
        """
        for message_id in self.msgids.keys():
            id, msg = maildb.msg_by_id(self.msgids[message_id])
            self.assertEqual(msg['message-id'], message_id)
            self.assertEqual(id, self.msgids[message_id])

    def test_ids(self):
        """
        Verify that items retrieved by message_id have the correct Message-ID.
        """
        for id in self.message_ids.keys():
            id1, msg = maildb.msg_by_message_id(self.message_ids[id])
            self.assertEqual(msg['message-id'], self.message_ids[id])
            self.assertEqual(id, id1)

    def test_dates(self):
        """
        Verify that retrieving records between the minimum and maximum dates
        returns an appropriate number of records, and that each separate day
        shows one email for each sender.
        """
        mind = min(self.msgdates)
        mindate = datetime.date(mind.year, mind.month, mind.day)
        maxd = max(self.msgdates)
        maxdate = datetime.date(maxd.year, maxd.month, maxd.day)
        self.assertEqual(self.rowcount,
                         len(maildb.msgs~~_by_date~~(mindate=mindate,
                                                maxdate=maxdate)))

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

Of course, we expected all tests to pass. And the **mlist1.py** program that we wrote earlier still functions perfectly, even though new columns have been added to the table since last you ran it:

OBSERVE:

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.892s

OK
```

Adding A GUI

Our tests give us some confidence that our email storage library is sound. How difficult would it be to build a graphical user interface to use with it? Not too difficult if we use a basic layout to prototype the program.

In earlier lessons, we used the **tkinter** grid layout to produce quick interface layouts. This is fine—so long as when the final interface is produced, the widgets that matter (the ones used by the methods) keep the same names.

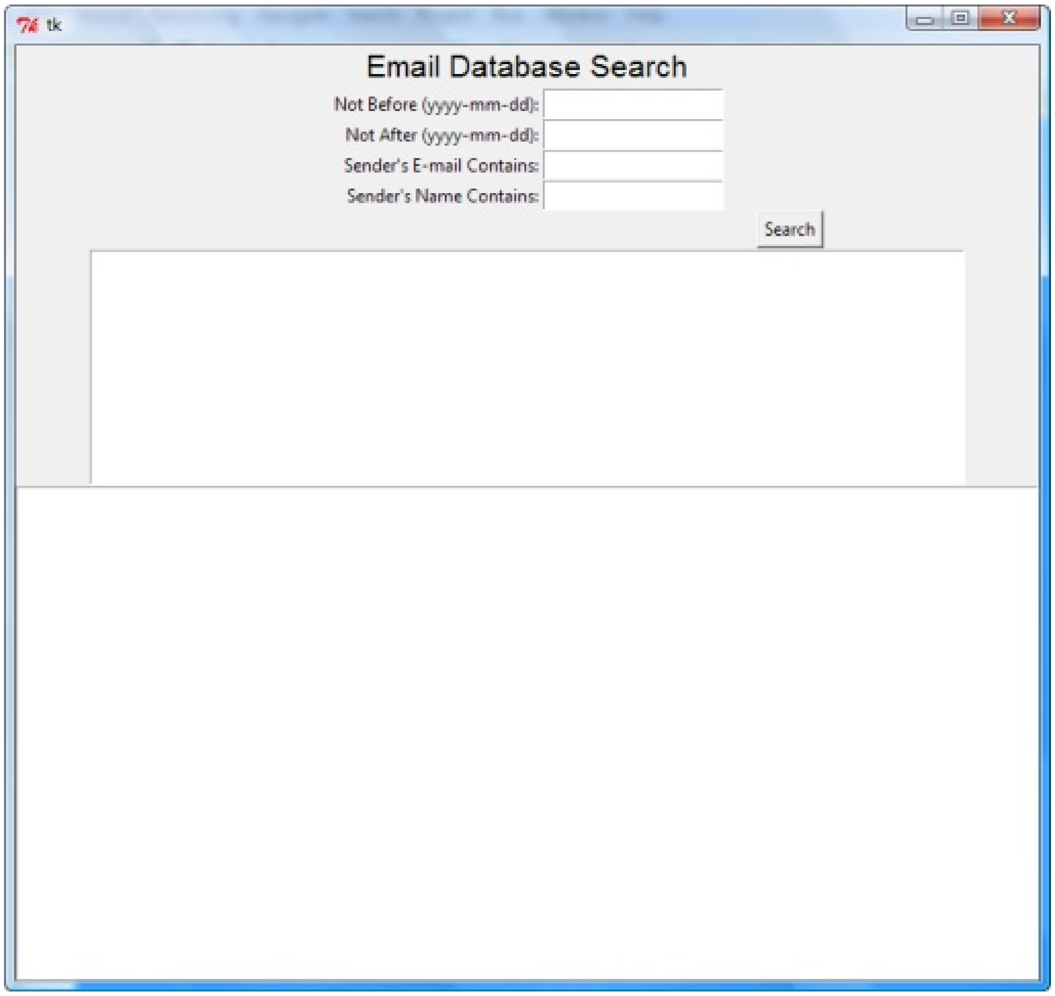
This particular application offers four search field entries: two for the minimum and maximum dates, one for the email address, and one for the name. We'll place these with appropriate labels on a four-by-two grid, with the labels right-justified and the entry widgets left-justified. We'll add a button to trigger the search to the second column in the fifth row, and the final two rows will hold a listbox and a text widget.

In the **EmailSearch/src** folder, create **mailgui.py** as shown:

CODE TO TYPE:

```
from tkinter import *
from maildb import msgs
import datetime

class Application(Frame):

    def __init__(self, master=None):
        """
        Establish the window structure, leaving some widgets accessible
        as app instance variables.
        """
        Frame.__init__(self, master)
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0, weight=1)
        self.grid(sticky=W+E+N+S)
        l0 = Label(self, text="Email Database Search", font=("Helvetica", 16))
        l0.grid(row=0, column=1, columnspan=2)
        l1 = Label(self, text="Not Before (yyyy-mm-dd):")
        l1.grid(row=1, column=1, sticky=E+N+S)
        self.mindate = Entry(self)
        self.mindate.grid(row=1, column=2, sticky=W+N+S)
        l2 = Label(self, text="Not After (yyyy-mm-dd):")
        l2.grid(row=2, column=1, sticky=E+N+S)
        self.maxdate = Entry(self)
        self.maxdate.grid(row=2, column=2, sticky=W+N+S)
        l3 = Label(self, text="Sender's E-mail Contains:")
        l3.grid(row=3, column=1, sticky=E+N+S)
        self.addsearch = Entry(self)
        self.addsearch.grid(row=3, column=2, sticky=W+N+S)
        l4 = Label(self, text="Sender's Name Contains:")
        l4.grid(row=4, column=1, sticky=E+N+S)
        self.namesearch = Entry(self)
        self.namesearch.grid(row=4, column=2, sticky=W+N+S)
        button = Button(self, text="Search")
        button.grid(row=5, column=2)
        self.msgsubs = Listbox(self, height=10, width=100)
        self.msgsubs.grid(row=8, column=1, columnspan=2)
        self.message = Text(self, width=100)
        self.message.grid(row=9, column=1, columnspan=2)

if __name__ == "__main__":

    root = Tk()
    app = Application(master=root)
    app.mainloop()
```

When you run this code, you see a GUI that looks like this—as promised, ugly but functional:



With the interface rendering properly as a window on the screen, now we need to plug in the "works." First, we'll add a search routine to run when the **Search** button is clicked. It should perform a search and populate the Listbox with the subject lines of each message.

The **maildb.msgs** search function does not require all arguments, but we want to be able to search on all of them, we'll provide them all. We'll arrange for the value **None** to be presented whenever the user's Entry is empty.

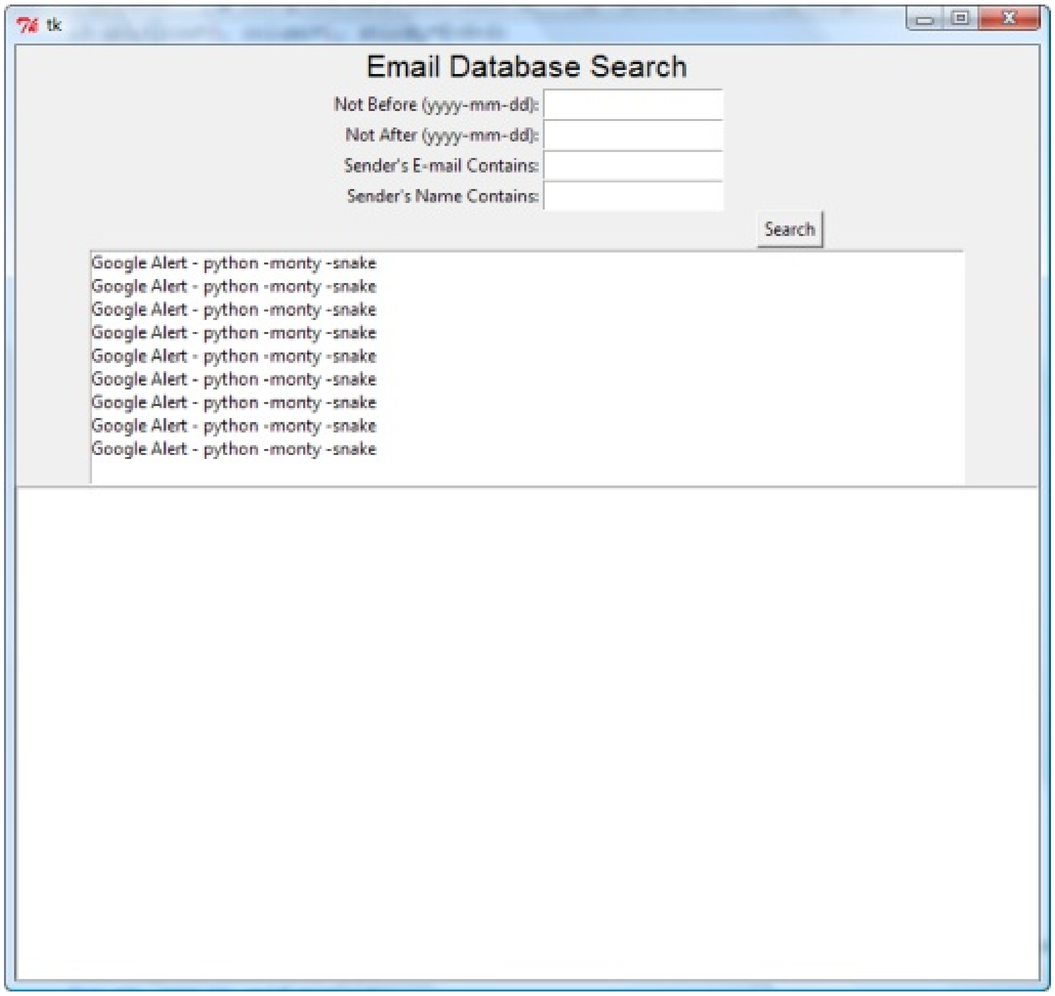
Dates are just a little trickier. We'll add a simple conversion function, and require that the user enters dates as "YYYY-MM-DD." It isn't particularly user-friendly to require such closely-formatted entries, but we can improve that later if necessary. The function converts those strings into a **datetime.date** object for passing to **maildb.msgs()**.

The main addition is the **search_mail()** method, which does all the necessary preparation and finally calls **maildb.msgs()** to retrieve the specified messages and display the subject header value of each in a Listbox. We trigger the instance's **search_mail()** method by adding it as the **command** configuration parameter to the Button's creation. The **search_mail()** method is also called at startup, before the window is displayed. Modify **mailgui.py** as shown:

CODE TO TYPE:

```
from tkinter import *
from maildb import msgs
import datetime

def get_date(s):
    """
    Assumes a date of form yyyy-mm-dd, returns a corresponding datetime.date.
    """
    syear = s[:4]
    smonth = s[5:7]
    sday = s[8:]
    return datetime.date(int(syear), int(smonth), int(sday))

class Application(Frame):

    def __init__(self, master=None):
        """
        Establish the window structure, leaving some widgets accessible
        as app instance variables. Connect button clicks to search_mail
        method.
        """
        Frame.__init__(self, master)
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0, weight=1)
        self.grid(sticky=W+E+N+S)
        l0 = Label(self, text="Email Database Search", font=("Helvetica", 16))
        l0.grid(row=0, column=1, columnspan=2)
        l1 = Label(self, text="Not Before (yyyy-mm-dd):")
        l1.grid(row=1, column=1, sticky=E+N+S)
        self.mindate = Entry(self)
        self.mindate.grid(row=1, column=2, sticky=W+N+S)
        l2 = Label(self, text="Not After (yyyy-mm-dd):")
        l2.grid(row=2, column=1, sticky=E+N+S)
        self.maxdate = Entry(self)
        self.maxdate.grid(row=2, column=2, sticky=W+N+S)
        l3 = Label(self, text="Sender's E-mail Contains:")
        l3.grid(row=3, column=1, sticky=E+N+S)
        self.addsearch = Entry(self)
        self.addsearch.grid(row=3, column=2, sticky=W+N+S)
        l4 = Label(self, text="Sender's Name Contains:")
        l4.grid(row=4, column=1, sticky=E+N+S)
        self.namesearch = Entry(self)
        self.namesearch.grid(row=4, column=2, sticky=W+N+S)
        button = Button(self, text="Search", command=self.search_mail)
        button.grid(row=5, column=2)
        self.msgsubs = Listbox(self, height=10, width=100)
        self.msgsubs.grid(row=8, column=1, columnspan=2)
        self.message = Text(self, width=100)
        self.message.grid(row=9, column=1, columnspan=2)

    def search_mail(self):
        """
        Take the database search parameters provided by the user
        (trying to make sense of the dates) and select the appropriate
        messages from the database, displaying the subject lines of the
        messages in a scrolling selection list.
        """
        mindate = self.mindate.get()
        if not mindate:
            mindate = None
        else:
            mindate = get_date(mindate)
        maxdate = self.maxdate.get()
        if not maxdate:
            maxdate = None
        else:
            maxdate = get_date(maxdate)
        addsearch = self.addsearch.get()
        if not addsearch:
            addsearch = None
        namesearch = self.namesearch.get()
        if not namesearch:
            namesearch = None
        self.msglist = msgs(mindate=mindate, maxdate=maxdate, addsearch=addsearc
h, namesearch=namesearch)
        self.msgsubs.delete(0, END)
        for pk, msg in self.msglist:
            self.msgsubs.insert(END, msg['subject'])

if __name__ == "__main__":

    root = Tk()
    app = Application(master=root)
    app.search_mail()
    app.mainloop()
```

Now we have a program that will list the subject lines of the messages that meet the search criteria. By default, you'll see whatever content is in the database (which is usually whatever was left by the last test in the messages table). So the window looks more or less the same when you run it as it did before, except that you see messages listed in the Listbox.



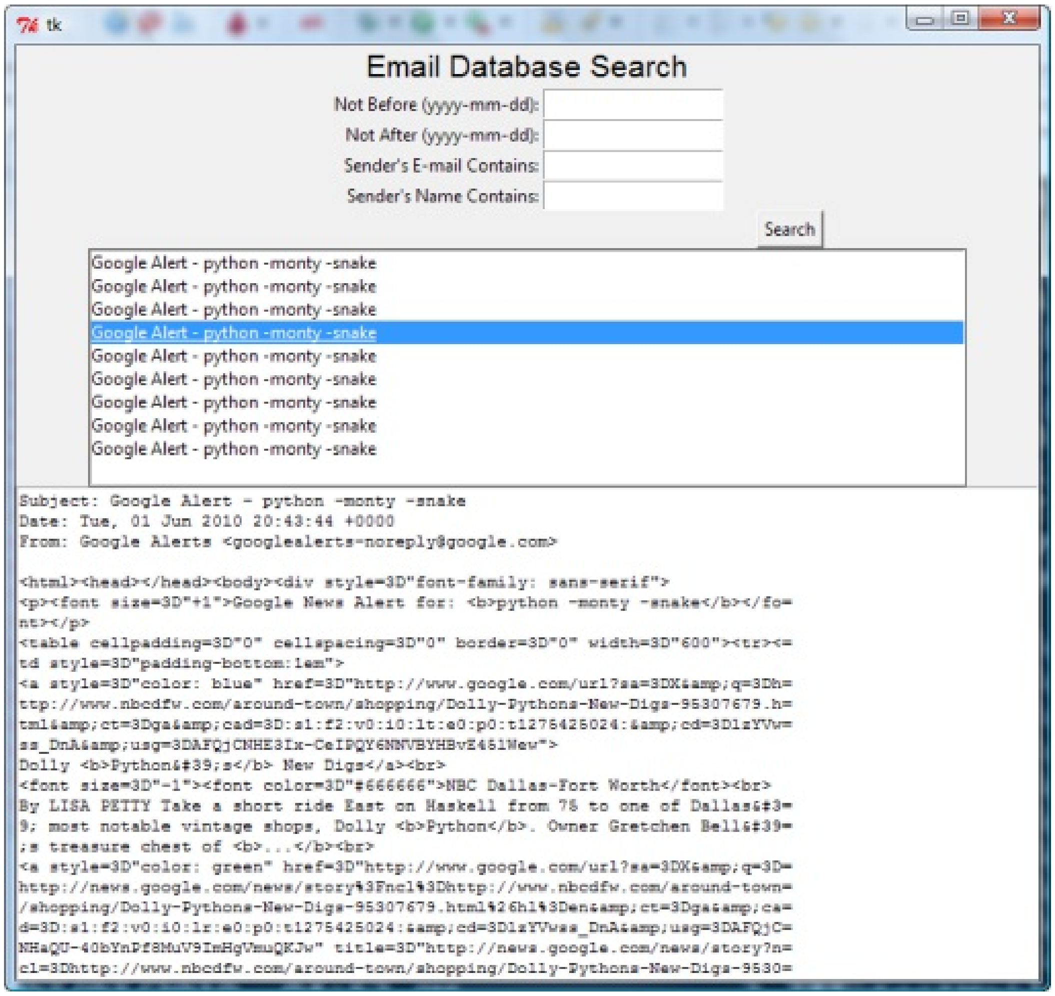
```
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
```

The final step is to connect a double-click on a Listbox entry to display the content of that message in the Text widget at the bottom of the window. Again, the code changes are fairly straightforward. The required double-click event is bound to the new **display_mail()** method, and the method extracts the selection from the Listbox and deletes any existing content from the Text widget. Then it inserts up to three headers, followed by a blank line and the body of the messages (unless it happens to be a multipart message—those are a little trickier to handle). Modify **mailgui.py** as shown:

CODE TO TYPE:

```
from tkinter import *
from maildb import msgs
import datetime

def get_date(s):
    """
    Assumes a date of form yyyy-mm-dd, returns a corresponding datetime.date.
    """
    syear = s[:4]
    smonth = s[5:7]
    sday = s[8:]
    return datetime.date(int(syear), int(smonth), int(sday))

class Application(Frame):

    def __init__(self, master=None):
        """
        Establish the window structure, leaving some widgets accessible
        as app instance variables. Connect button clicks to search_mail
        method and subject double-clicks to display_mail method.
        """
        Frame.__init__(self, master)
        self.master.rowconfigure(0, weight=1)
        self.master.columnconfigure(0, weight=1)
        self.grid(sticky=W+E+N+S)
        l0 = Label(self, text="Email Database Search", font=("Helvetica", 16))
        l0.grid(row=0, column=1, columnspan=2)
        l1 = Label(self, text="Not Before (yyyy-mm-dd):")
        l1.grid(row=1, column=1, sticky=E+N+S)
        self.mindate = Entry(self)
        self.mindate.grid(row=1, column=2, sticky=W+N+S)
        l2 = Label(self, text="Not After (yyyy-mm-dd):")
        l2.grid(row=2, column=1, sticky=E+N+S)
        self.maxdate = Entry(self)
        self.maxdate.grid(row=2, column=2, sticky=W+N+S)
        l3 = Label(self, text="Sender's E-mail Contains:")
        l3.grid(row=3, column=1, sticky=E+N+S)
        self.addsearch = Entry(self)
        self.addsearch.grid(row=3, column=2, sticky=W+N+S)
        l4 = Label(self, text="Sender's Name Contains:")
        l4.grid(row=4, column=1, sticky=E+N+S)
        self.namesearch = Entry(self)
        self.namesearch.grid(row=4, column=2, sticky=W+N+S)
        button = Button(self, text="Search", command=self.search_mail)
        button.grid(row=5, column=2)
        self.msgsubs = Listbox(self, height=10, width=100)
        self.msgsubs.grid(row=8, column=1, columnspan=2)
        self.msgsubs.bind("<Double-Button-1>", self.display_mail)
        self.message = Text(self, width=100)
        self.message.grid(row=9, column=1, columnspan=2)

    def search_mail(self):
        """
        Take the database search parameters provided by the user
        (trying to make sense of the dates) and select the appropriate
        messages from the database, displaying the subject lines of the
        messages in a scrolling selection list.
        """
        mindate = self.mindate.get()
        if not mindate:
            mindate = None
        else:
            mindate = get_date(mindate)
        maxdate = self.maxdate.get()
        if not maxdate:
            maxdate = None
        else:
            maxdate = get_date(maxdate)
        addsearch = self.addsearch.get()
        if not addsearch:
            addsearch = None
        namesearch = self.namesearch.get()
        if not namesearch:
            namesearch = None
        self.msglist = msgs(mindate=mindate, maxdate=maxdate, addsearch=addsearc
h, namesearch=namesearch)
        self.msgsubs.delete(0, END)
        for pk, msg in self.msglist:
            self.msgsubs.insert(END, msg['subject'])

    def display_mail(self, event):
        """
        Display the message corresponding to the subject line that the
        user just clicked on.
        """
        indexes = self.msgsubs.curselection()
        if len(indexes) != 1:
            return
        self.message.delete(1.0, END)
        pk, msg = self.msglist[int(indexes[0])]
        for header_name in "Subject", "Date", "From":
            hdr = msg[header_name]
            if hdr:
                self.message.insert(INSERT, "{0}: {1}\n".format(header_name, hdr
))
        self.message.insert(END, "\n")
        if msg.is_multipart():
            self.message.insert(END, "MULTIPART MESSAGE - SORRY!")
        self.message.insert(END, msg.get_payload())

if __name__ == "__main__":

    root = Tk()
    app = Application(master=root)
    app.search_mail()
    app.mainloop()
```

When you run this modified code, you see the final (but not necessarily complete) form of our GUI-based mail retrieval program. It searches messages by date range, sender name, and email address, and allows you to view any message in the search results by double-clicking the message subject. This sort of code might be considered "alpha quality"—it can be released for testing purposes, but it's not quite ready for prime time.



```
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
Google Alert - python -monty -snake
```

```
Subject: Google Alert - python -monty -snake
Date: Tue, 01 Jun 2010 20:43:44 +0000
From: Google Alerts <googlealerts-noreply@google.com>

<html><head></head><body><div style=3D"font-family: sans-serif">
<p><font size=3D"+1">Google News Alert for: <b>python -monty -snake</b></fo=
nt></p>
<table cellpadding=3D"0" cellspacing=3D"0" border=3D"0" width=3D"600"><tr><=
td style=3D"padding-bottom:1em">
<a style=3D"color: blue" href=3D"http://www.google.com/url?sa=3DX&amp;q=3Dh=
ttp://www.nbcdfw.com/around-town/shopping/Dolly-Pythons-New-Digs-95307679.h=
tml&amp;ct=3Dga&amp;cad=3D:s1:f2:v0:i0:lt:e0:p0:t1275425024:&amp;cd=3D1zYVw=
ss_DnA&amp;usg=3DAFQjCNHE3Ix-CeIPQY6NMVBYHBvE451New">
Dolly <b>Python&#39;s</b> New Digs</a><br>
<font size=3D"-1"><font color=3D"#666666">NBC Dallas-Fort Worth</font><br>
By LISA PETTY Take a short ride East on Haskell from 75 to one of Dallas&#3=
9; most notable vintage shops, Dolly <b>Python</b>. Owner Gretchen Bell&#39=
;s treasure chest of <b>...</b><br>
<a style=3D"color: green" href=3D"http://www.google.com/url?sa=3DX&amp;q=3D=
http://news.google.com/news/story%3Fncl%3Dhttp://www.nbcdfw.com/around-town=
/shopping/Dolly-Pythons-New-Digs-95307679.html%26hl%3Den&amp;ct=3Dga&amp;ca=
d=3D:s1:f2:v0:i0:lr:e0:p0:t1275425024:&amp;cd=3D1zYVwss_DnA&amp;usg=3DAFQjC=
NHaQU-40bYnPf8MuV9ImHgVmuQKJw" title=3D"http://news.google.com/news/story?n=
cl=3Dhttp://www.nbcdfw.com/around-town/shopping/Dolly-Pythons-New-Digs-9530=
```

The appearance of the interface could be improved, but the program's basic design is sound. The program is constructed plainly, and we can see how to extend it in various ways.

For example, if you wanted to add subject search features, it's pretty clear that you'd need to add an **msgSubject** column to the message table and therefore to the logic of **maildb.store()**. The interface to **maildb.msgs()** would need to be augmented by a **subjectsearch** argument, and the GUI would need to add another Entry element to capture the user's search string. Fortunately, this program is logically organized, and you should be able to proceed with confidence.

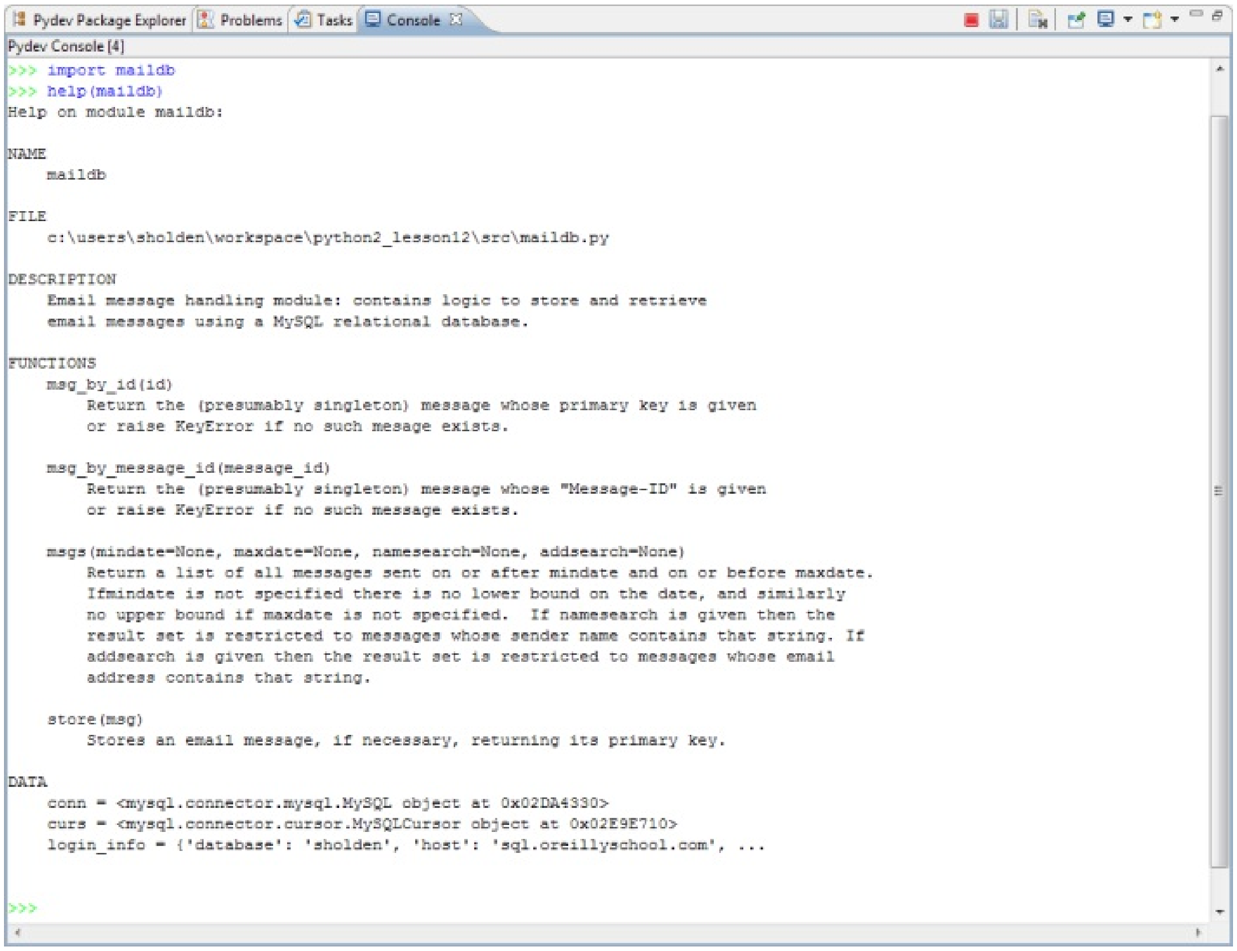
Documentation

Open a new pydev console, select the **Console** tab, and maximize the console so that you can see the output. Enter the commands below as shown:

INTERACTIVE SESSION:

```
>>> import maildb
>>> help(maildb)
```

The Python help system uses all of the docstrings you've put into your code to produce a brief description of your **maildb** module.



```
>>> import maildb
>>> help(maildb)
Help on module maildb:

NAME
    maildb

FILE
    c:\users\sholden\workspace\python2_lesson12\src\maildb.py

DESCRIPTION
    Email message handling module: contains logic to store and retrieve
    email messages using a MySQL relational database.

FUNCTIONS
    msg_by_id(id)
        Return the (presumably singleton) message whose primary key is given
        or raise KeyError if no such mesage exists.

    msg_by_message_id(message_id)
        Return the (presumably singleton) message whose "Message-ID" is given
        or raise KeyError if no such message exists.

    msgs(mindate=None, maxdate=None, namesearch=None, addsearch=None)
        Return a list of all messages sent on or after mindate and on or before maxdate.
        Ifmindate is not specified there is no lower bound on the date, and similarly
        no upper bound if maxdate is not specified.  If namesearch is given then the
        result set is restricted to messages whose sender name contains that string. If
        addsearch is given then the result set is restricted to messages whose email
        address contains that string.

    store(msg)
        Stores an email message, if necessary, returning its primary key.

DATA
    conn = <mysql.connector.mysql.MySQL object at 0x02DA4330>
    curs = <mysql.connector.cursor.MySQLCursor object at 0x02E9E710>
    login_info = {'database': 'sholden', 'host': 'sql.oreillyschool.com', ...


>>>
```

Congratulations! Your hard work is really paying off. You've powered through all of the challenges we've thrown at you and arrived at the finish line of this second O'Reilly School of Technology Python course. Your command of the language is astounding! You can integrate databases and graphical user interfaces, and you're prepared to explore the bigger Python landscape. Now let's dazzle your instructor and put those skills to work in your final project! It's been a real pleasure working with you. See you in the next course!