Python 4: Advanced Python

Going Further with Functions

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

Course Objectives

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

- extend Python code functionality through inheritance, complex delegation, and recursive composition.
- publish, subscribe, and optimize your code.
- create advanced class decorators and generators in Python.
- demonstrate knowledge of Python introspection.
- apply multi-threading and mult-processing to Python development.
- manage arithmetic contexts and memory mapping.
- demonstrate understanding of the Python community, conferences, and job market.
- develop a multi-processing solution to a significant data processing problem.

This course builds on your existing Python knowledge, incorporating further object-oriented design principles and techniques with the intention of rounding out your skill set. Techniques like recursion, composition, and delegation are explained and put into practice through the ever-present test-driven practical work.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* 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.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

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 also include instructive comments that you don't need to type.
```

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've already taken the OST course on **Introduction to Python**, **Getting More Out of Python** and/or **The Python Environment**, 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 use 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.

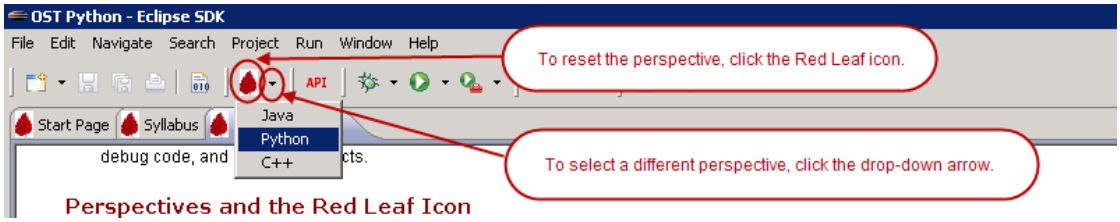
Perspectives and the Red Leaf Icon

The Ellipse Plug-in for Eclipse was developed by OST. It 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.

To use the Red Leaf icon to:

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

  For this course, select **Python**:



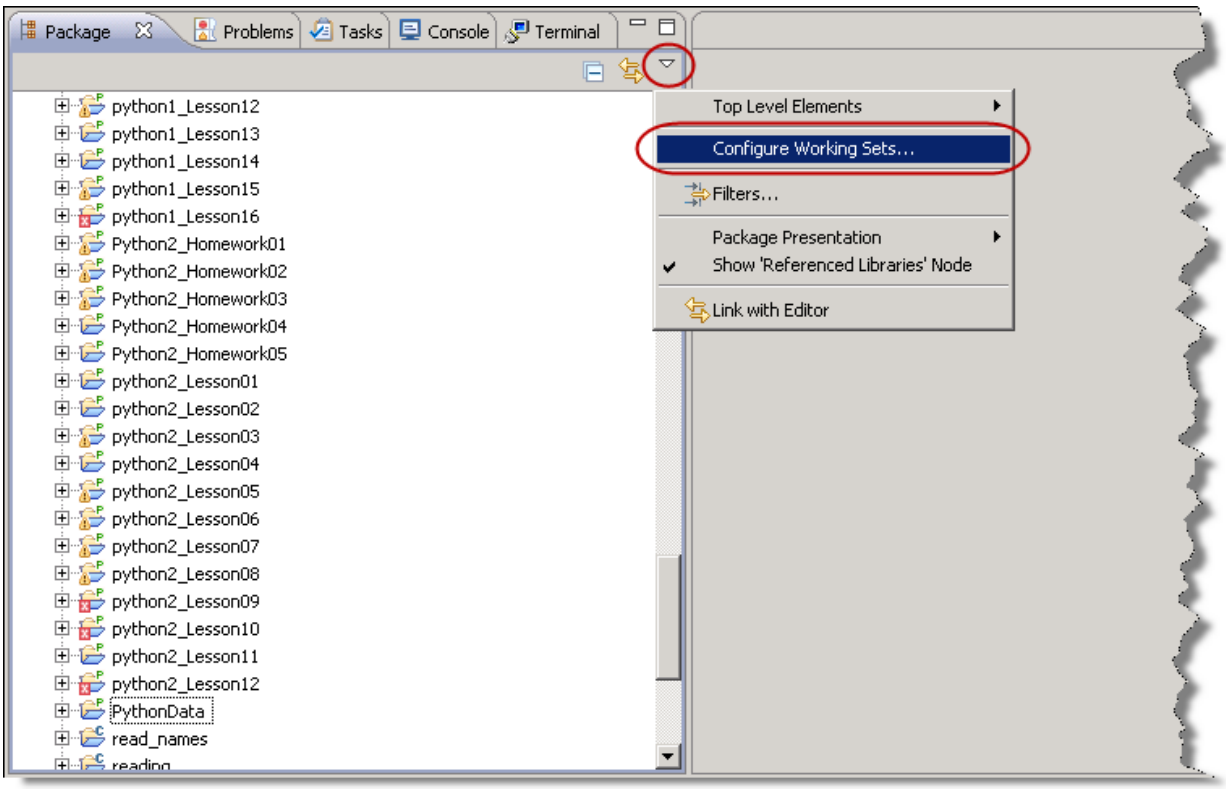
Working Sets

In this course, we'll use *working sets*. All projects created in Eclipse exist in the workspace directory of your account on our server. As you create projects throughout the 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.

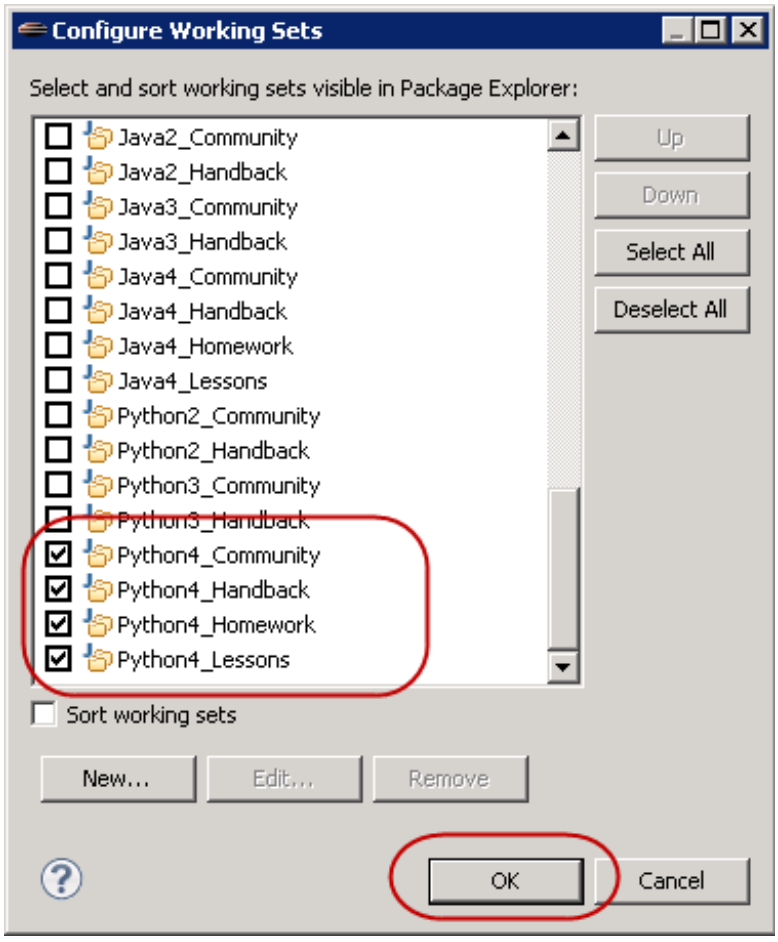
A working set is a convenient way to group related items together. You can assign a project to one or more working sets. In some cases, like the Python extension to Eclipse, new projects are created in a catch-all "Other Projects" working set. To organize your work better, we'll have you assign your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

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

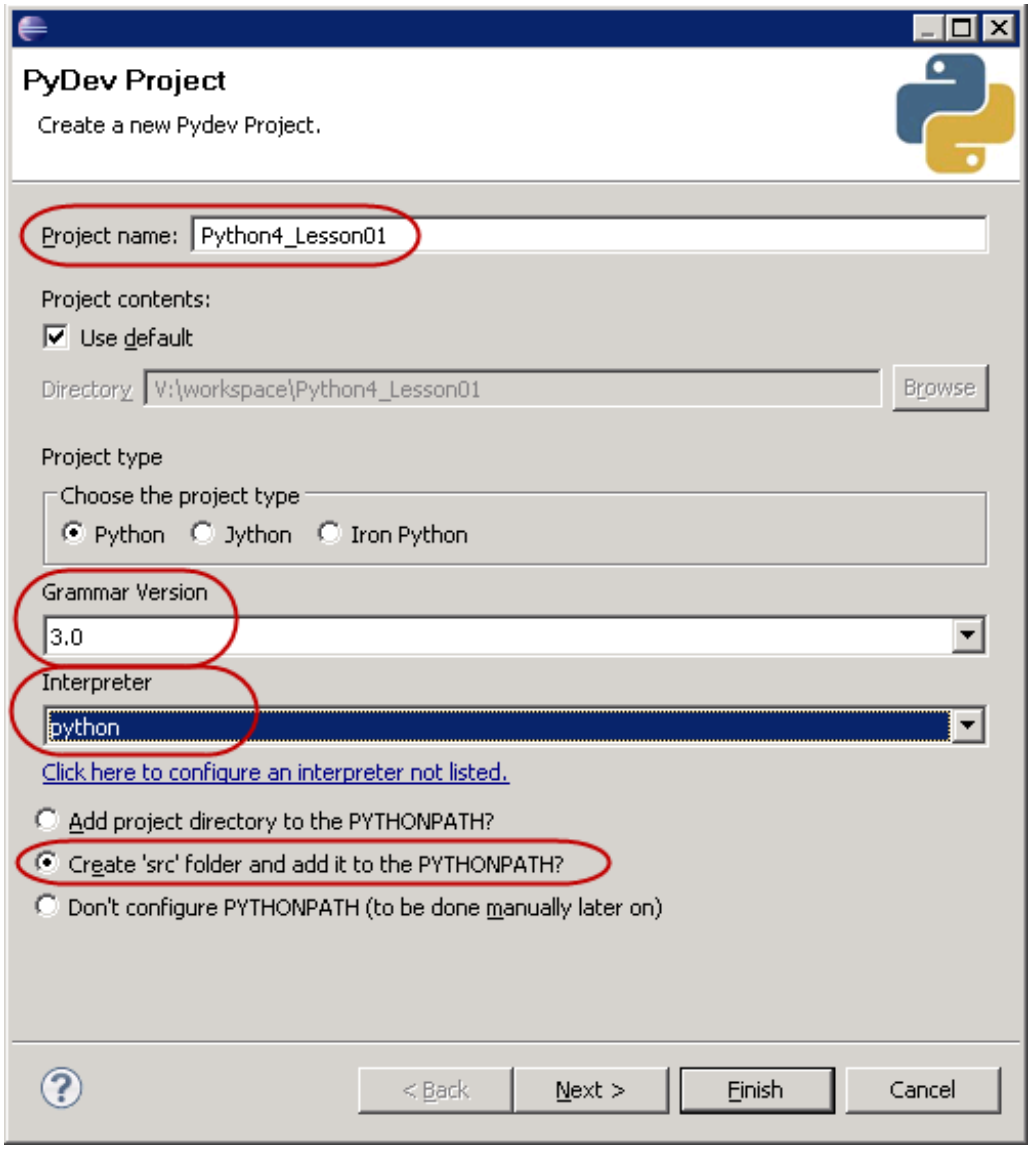
For this course, we'll display only the working sets you need. In the upper-right corner of the Package Explorer panel, click the downward arrow and select **Configure Working Sets**:



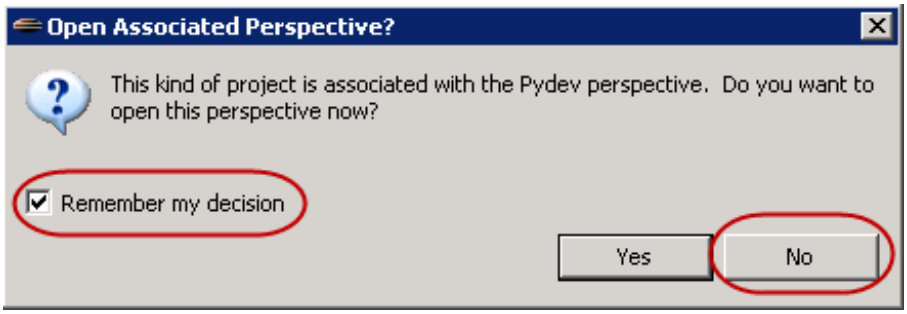
Select the **Other Projects** working set as well as the ones that begin with "Python4," then click **OK**:



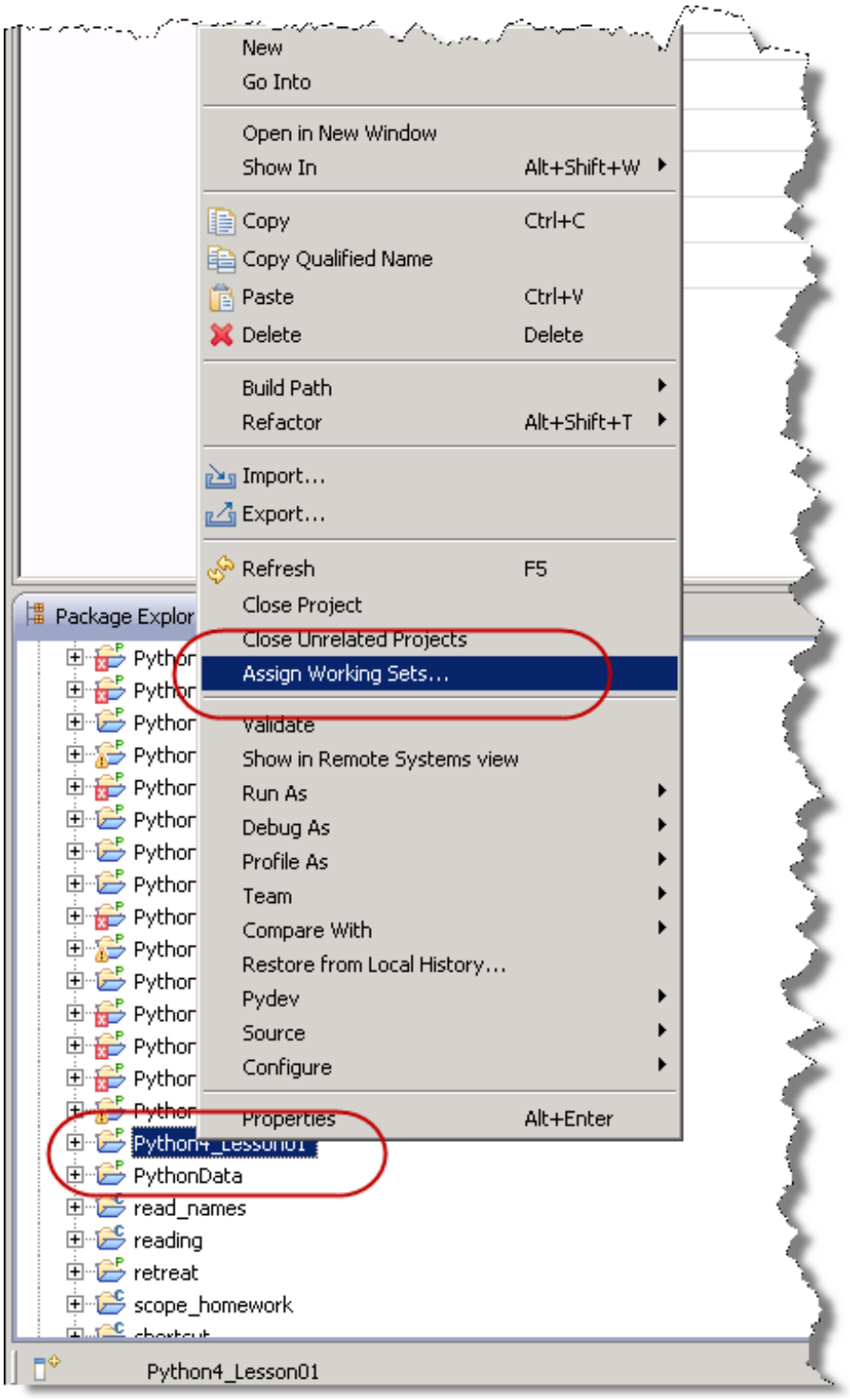
Let's create a project to store our programs for this lesson. Select **File | New | Pydev Project**, and enter the information as shown:



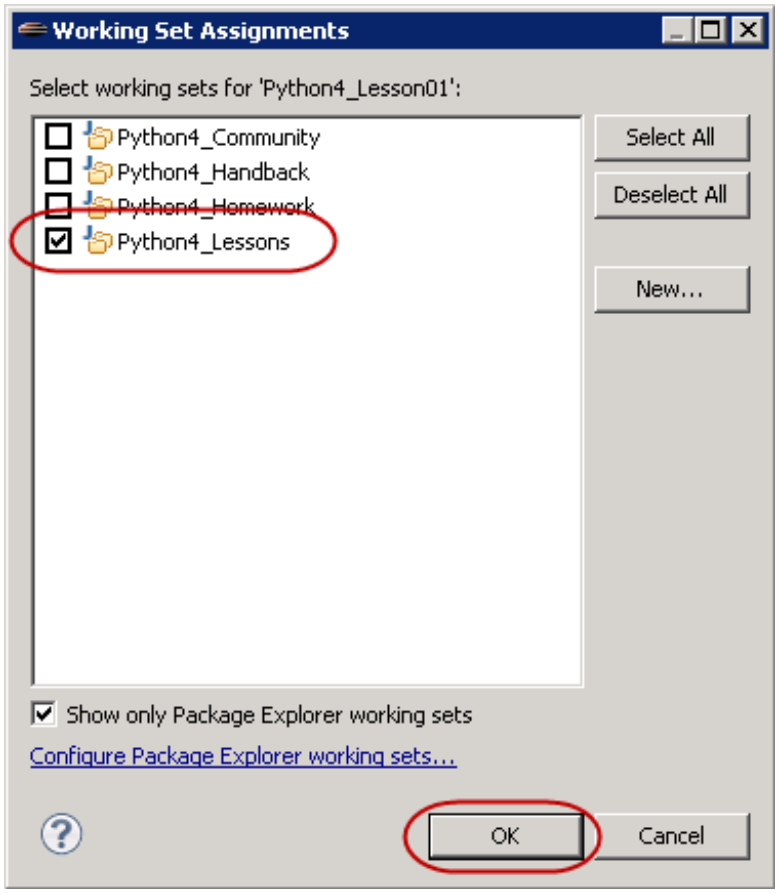
Click **Finish**. When asked if you want to open the associated perspective, check the **Remember my decision** box and click **No**:



By default, the new project is added to the Other Projects working set. Find **Python4_Lesson01** there, right-click it, and select **Assign Working Sets...** as shown:



Select the **Python4_Lessons** working set and click **OK**:



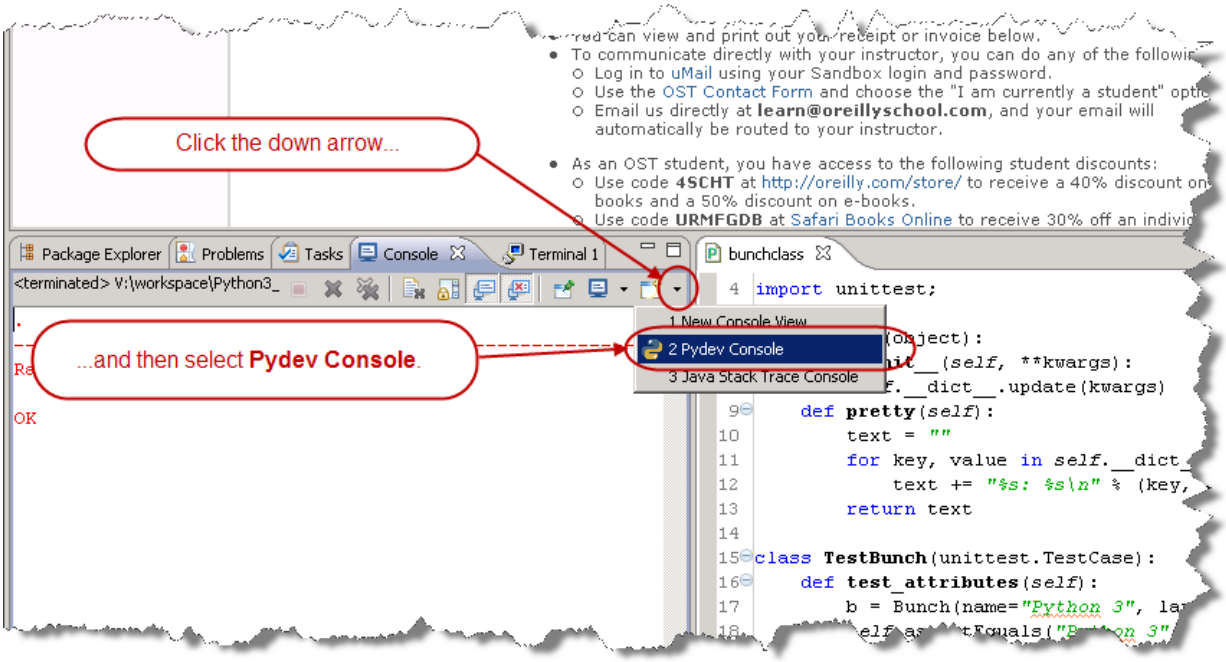
In the next section, we'll get to enter some Python code and run it!

Functions Are Objects

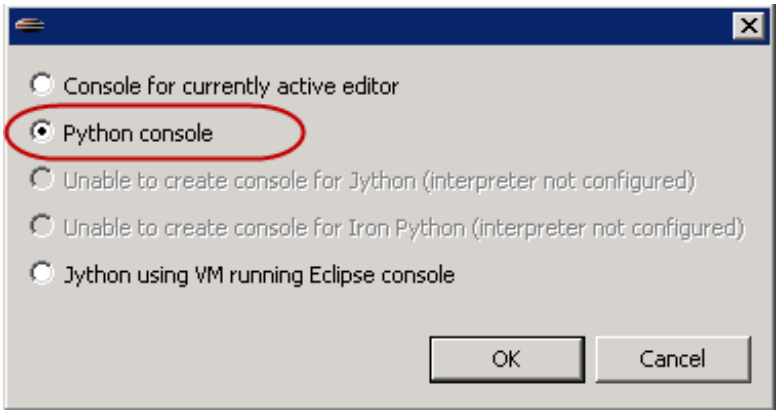
Everything in Python is an object, but unlike most objects in Python, function objects are not created by calling a class. Instead you use the **def** statement, which causes the interpreter to compile the indented suite that comprises the function body and bind the compiled code object to the function's name in the current local namespace.

Function Attributes

Like any object in Python, functions have a particular type; and like with any object in Python, you can examine a function's namespace with the **dir()** function. Let's open a new interactive session. Select the **Console** tab, click the down arrow and select **Pydev console**:



In the dialog that appears, select **Python console**:



Then, type the commands shown:

INTERACTIVE SESSION:

```
>>> def g(x):
...     return x*x
...
>>> g
<function g at 0x100572490>
>>> type(g)
<class 'function'>
>>> dir(g)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__get__', '__getattribute__', '__globals__', '__gt__', '__hash__',
'__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
>>>
```

Note Keep this interactive session open throughout this lesson.

While this tells you what attributes function objects possess, it does not make it very clear which of them are

unique to functions. A good Python programmer like you needs to be able to think of a way to discover the attributes of function that aren't also attributes of the base object, **object**.

Think about it for a minute. Here's a hint: think about sets.

You may remember that the set() function produces a set when applied to any iterable (which includes lists: the dir() function returns a list). You may also remember that sets implement a subtraction operation: if *a* and *b* are sets, then *a-b* is the set of items in *a* that are not also in *b*. Continue the interactive sesson as shown:

INTERACTIVE SESSION:

```
>>> def f(x):
...     return x
...
>>> function_attrs = set(dir(f))
>>> object_attrs = set(dir(object))
>>> function_attrs -= object_attrs
>>> from pprint import pprint
>>> pprint(sorted(function_attrs))
['__annotations__',
 '__call__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__dict__',
 '__get__',
 '__globals__',
 '__kwdefaults__',
 '__module__',
 '__name__']
>>>
```

At this stage in your Python programming career, you don't need to worry about most of these, but there's certainly no harm in learning what they do. Some of the features they offer are very advanced. You can read more about them in the official Python documentation. You can learn a lot by working on an interactive terminal session and by reading the documentation.

Function and Method Calls

The __call__() method is interesting—its name implies that it has something to do with function calling, and this is correct. The interpreter calls any callable object by making use of its __call__() method. You can actually call this method directly if you want to; it's exactly the same as calling the function directly.

INTERACTIVE SESSION:

```
>>> def f1(x):
...     print("f1({}) called".format(x))
...     return x
...
>>> f1.__call__(23) # should be equivalent to f1(23)
f1(23) called
23
>>>
```

You can define your own classes to include a __call__() method, and if you do, the instances you create from that class will be callable directly, just like functions. This is a fairly general mechanism that illustrates a Python equivalence you haven't observed yet:

f(*args, **kw)

≡

f.__call__(*args, **kw)

Give it a try. Create a class with instances that are callable. Then verify that you can call the instances:

INTERACTIVE SESSION:

```
>>> class Func:
...     def __call__(self, arg):
...         print("%r(%r) called" % (self, arg))
...         return arg
...
>>> f2 = Func()
>>> f2
<__main__.Func object at 0x100569dd0>
>>> f2("Danny")
<__main__.Func object at 0x100569dd0>('Danny') called
'Danny'
>>>
```

As we've seen, when you define a __call__() method on the class, you can call its instances. These calls result in the activation of the __call__() method, with the instance provided (as always on a method call) as the first argument, followed by the positional and keyword arguments that were passed to the instance call. Methods are normally defined on a class. While it is possible to bind callable objects to names in an instance's namespace, the interpreter does *not* treat it as a true method, and as such, it does not add the instance as a first argument. So, callables in the instance's __dict__ are called with only the arguments present on the call line—no instance is implicitly added as a first argument.

Note

The so-called "magic" methods (those with names that begin and end with a double underscore) are *never* looked for on the instance—the interpreter goes straight to the classes for these methods. So even when the instance's __dict__ contains the key "__call__", it is ignored and the class's __call__() method is activated.

Let's continue our console session:

INTERACTIVE SESSION:

```
>>> def userfunc(arg):
...     print("Userfunc called: ", arg)
...
>>> f2.regular = userfunc
>>> f2.regular("Instance")
Userfunc called: Instance
>>> f2.__call__ = userfunc
>>> f2("Hopeful")
<__main__.Func object at 0x100569dd0>('Hopeful') called
'Hopeful'
```

Since all callables have a __call__() method, and the __call__() method is callable, you might wonder whether it too has a __call__() method. The answer is yes, it does (and so does that __call__() method, and so on...):

INTERACTIVE SESSION:

```
>>> "__call__" in dir(f2.__call__)
True
>>> f2.__call__("Audrey")
Userfunc called: Audrey
>>> f2.__call__.__call__("Audrey")
Userfunc called: Audrey
>>> f2.__call__.__call__.__call__("Audrey")
Userfunc called: Audrey
>>>
```

Function Composition

Because functions are first-class objects, they can be passed as arguments to other functions, and such. If *f* and *g* are functions, then mathematicians defined the composition *f * g* of those two functions by saying that *(f * g)(x) = f(g(x))*. In other words, the composition of two functions is a new function, that behaves the same as applying the first function to the output of the second.

Suppose you were given two functions; could you construct their composition? Of course you could! For example, you could write a function that takes two functions as arguments, then internally defines a function that calls the first on the result of the second. Then the compose function returns that function. It's actually almost easier to write the function than it is to describe it:

INTERACTIVE SESSION:

```
>>> def compose(g, h):
...     def anon(x):
...         return g(h(x))
...     return anon
...
>>> f3 = compose(f1, f2)
>>> f3("Shillalegh")
<__main__.Func object at 0x100569dd0>('Shillalegh') called
f1('Shillalegh') called
'Shillalegh'
```

While it's pretty straightforward to compose functions this way, a mathematician would find it much more natural to compose the functions with a multiplication operator (the asterisk*). Unfortunately, an attempt to multiply two functions together is doomed to fail, as Python functions have not been designed to be multiplied. If we could add a __mul__() method to our functions, we might stand a chance, but as we've seen, this is not possible with function instances, and the class of functions is a built-in object written in C: impossible to change and difficult from which to inherit. Even when you do subclass the function type, how would you create instances? The def statement will always create regular functions.

While you may not be able to subclass the function object, you *do* know how to create object classes with callable instances. Using this technique, you could create a class with instances that act as proxies for the functions. This class could define a __mul__() method, which would take another similar class as an argument and return the composition of the two proxied functions. This is typical of the way that Python allows you to "hook" into its workings to achieve a result that is simpler to use.

In your **Python4_Lesson01/src** folder, create a program called **composable.py** as shown below:

CODE TO TYPE:

```
"""
composable.py: defines a composable function class.
"""
class Composable:
    def __init__(self, f):
        "Store reference to proxied function."
        self.func = f
    def __call__(self, *args, **kwargs):
        "Proxy the function, passing all arguments through."
        return self.func(*args, **kwargs)
    def __mul__(self, other):
        "Return the composition of proxied and another function."
        if type(other) is Composable:
            def anon(x):
                return self.func(other.func(x))
            return Composable(anon)
        raise TypeError("Illegal operands for multiplication")
    def __repr__(self):
        return "<Composable function {0} at 0x{1:X}>".format(
                            self.func.__name__, id(self))
```

Save and run it. (Remember how to run a Python program in OST's sandbox environment? Right-click in the editor window for the **testarray.py** file, and select **Run As | Python Run**.)

Note An alternative implementation of the __mul__() method might have used the statement **return self(other(x))**. Do you think that this would have been a better implementation? Why or why not?

You will need tests, of course. So you should also create a program called **test_composable.py** that reads as follows.

CODE TO TYPE:

```
"""
test_composable.py" performs simple tests of composable functions.
"""
import unittest
from composable import Composable

def reverse(s):
    "Reverses a string using negative-stride sequencing."
    return s[::-1]

def square(x):
    "Multiplies a number by itself."
    return x*x

class ComposableTestCase(unittest.TestCase):

    def test_inverse(self):
        reverser = Composable(reverse)
        nulltran = reverser * reverser
        for s in "", "a", "0123456789", "abcdefghijklmnopqrstuvwxyz":
            self.assertEquals(nulltran(s), s)

    def test_square(self):
        squarer = Composable(square)
        po4 = squarer * squarer
        for v, r in ((1, 1), (2, 16), (3, 81)):
            self.assertEqual(po4(v), r)

    def test_exceptions(self):
        fc = Composable(square)
        with self.assertRaises(TypeError):
            fc = fc * 3

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

The unit tests are relatively straightforward, simply comparing the expected results from known inputs with expected outputs. In older Python releases it could be difficult to find out which iteration of a loop had caused the assertion to fail, but with the improved error messages of newer releases this is much less of a problem: argument values for failing assertions are much better reported than previously.

The exception is tested by running the TestCase's assertRaises() method with a single argument (specifying the exception(s) that are expected and acceptable. Under these circumstances the method returns what is called a "context manager" that will catch and analyze any exceptions raised from the indented suite. (There is a broader treatment of context managers in a later lesson). When you run the test program you should see three successful tests.

Output from test_composable.py

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

OK
```

Once you get the idea of how this works, you'll soon realize that the __mul__() method could be extended to handle a regular function—in other words, as long as the operand to the left of the "*" is a Composable, the operand to the right would be either a Composable or a function. So the method can be extended slightly to make Composables more usable.

Let's go ahead and edit composable.py to allow composition with 'raw' functions:

CODE TO TYPE:

```
"""
composable.py: defines a composable function class.
"""
import types
class Composable:
    def __init__(self, f):
        "Store reference to proxied function."
        self.func = f
    def __call__(self, *args, **kwargs):
        "Proxy the function, passing all arguments through."
        return self.func(*args, **kwargs)
    def __mul__(self, other):
        "Return the composition of proxied and another function."
        if type(other) is Composable:
            def anon(x):
                return self.func(other.func(x))
            return Composable(anon)
        elif type(other) is types.FunctionType:
            def anon(x):
                return self.func(other(x))
            return Composable(anon)
        raise TypeError("Illegal operands for multiplication")
    def __repr__(self):
        return "<Composable function {0} at 0x{1:X}>".format(
                            self.func.__name__, id(self))
```

Now the updated __mul__() method does one thing if the right operand (other) is a Composable: it defines and returns a function that extracts the functions from both Composables, that is the composition of both of those functions. But if the right-side operator is a function (which you check for by using the types module, designed specifically to allow easy reference to the less usual Python types), then the function passed in as an argument can be used directly rather than having to be extracted from a Composable.

The tests need to be modified, but not as much as you might think. The simplest change is to have the test_square() method use a function as the right operand of its multiplications. This should not lose any testing capability, since the first two tests were formerly testing essentially the same things. A further exception test is also added to ensure that when the function is the left operand an exception is also raised.

CODE TO TYPE:

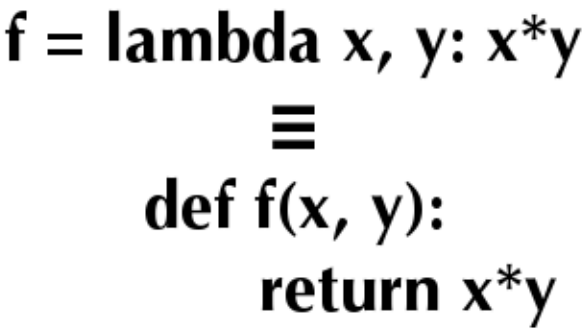
```
"""
test_composable.py" performs simple tests of composable functions.
"""
import unittest
from composable import Composable

def reverse(s):
    "Reverses a string using negative-stride sequencing."
    return s[::-1]

def square(x):
    "Multiplies a number by itself."
    return x*x

class ComposableTestCase(unittest.TestCase):

    def test_inverse(self):
        reverser = Composable(reverse)
        nulltran = reverser * reverser
        for s in "", "a", "0123456789", "abcdefghijklmnopqrstuvwxyz":
            self.assertEquals(nulltran(s), s)

    def test_square(self):
        squarer = Composable(square)
        po4 = squarer * squarer
        for v, r in ((1, 1), (2, 16), (3, 81)):
            self.assertEqual(po4(v), r)

    def test_exceptions(self):
        fc = Composable(square)
        with self.assertRaises(TypeError):
            fc = fc * 3
        with self.assertRaises(TypeError):
            fc = square * fc

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

A TypeError exception therefore *is* raised when you attempt to multiply a function by a Composable. The tests as modified should all succeed. If not, then debug your solution until they do, with your mentor's assistance if necessary.

The extensions you made to the Composable class in the last exercise made it more capable, but the last example shows that there are always wrinkles that you need to take care of to make your code as fully general as it can be. How far to go in adapting to all possible circumstances is a matter of judgment. Having a good set of tests at least ensures that the code is being exercised (it's also a good idea to employ *coverage testing*, to ensure that your tests don't leave any of the code unexecuted: this is not always as easy as you might think).

Lambdas: Anonymous Functions

Python also has a feature that allows you to define simple functions as an expression. The *lambda expression* is a way of expressing a function without having to use a def statement. Because it's an expression, there are limits to what you can do with a lambda. Some programmers use them frequently, but others prefer to define all of their functions. It's important for you to understand them, because you'll likely encounter them in other people's code.



While the equivalence above is not exact, it's close enough for all practical purposes. The keyword **lambda** is followed by the names of any parameters (all parameters to lambdas are positional) in a comma-separated list. A colon separates the parameters from the expression (normally referencing the parameters). The value of the expression will be returned from a call (you may need to restart the console, so you'll need to redefine some of the functions):

INTERACTIVE SESSION:

```
>>> def compose(g, h):
...     def anon(x):
...         return g(h(x))
...     return anon
...
>>>
>>> add1 = lambda x: x+1
>>> add1
<function <lambda> at 0x100582270>
>>> sqr = lambda x: x*x
>>> sqp1 = compose(sqr, add1)
>>> sqp1(5)
36
>>> type(add1)
<class 'function'>
>>>
```

It is relatively easy to write a lambda equivalent to the compose() function we created earlier—and it works as it would with any callable. The last result shows you that to the interpreter, lambda expressions are entirely equivalent to functions (lambda expressions and functions have the same type, "<class 'function'>").

Also, the lambda has no name (or more precisely: all lambdas have the *same* name). When you define a function with **def**, the interpreter stores the name from the def statement as its __name__ attribute. All lambdas have the same name, '<lambda>', when they are created. You can change that name by assignment to the attribute, but in general, if you're going to spend more than one line on a lambda, then you might as well just write a named function instead.

Finally, keep in mind that lambda is deliberately restricted to functions with bodies that comprise a single expression (which is implicitly what the lambda returns when called, with any argument values substituted for the parameters in the expression). Again, rather than writing expressions that continue over several lines, it would be better to write a named function (which, among other things, can be properly documented with docstrings). If you do wish to continue the expression over multiple lines, the best way to do that is to parenthesize the lambda expression. Do you think the parenthesized second version is an improvement? Think about that as you work through this interactive session:

INTERACTIVE SESSION:

```
>>> def f1(x):
...     print("f1({}) called".format(x))
...     return x
...
>>> class Func:
...     def __call__(self, arg):
...         print("%r(%r) called" % (self, arg))
...         return arg
...
>>> f2 = Func()
>>> ff = lambda f, g: lambda x: f(g(x))
>>> lam = ff(f1, f2)
>>> lam("Ebenezer")
<__main__.Func object at 0x10057a510>('Ebenezer') called
f1('Ebenezer') called
'Ebenezer'
>>>
>>> ff = lambda f, g: (lambda x:
...                         f(g(x)))
>>> lam = ff(f1, f2)
>>> lam("Ebenezer")
<__main__.Func object at 0x10057a510>('Ebenezer') called
f1('Ebenezer') called
'Ebenezer'
>>>
```

If you understand that last example, consider yourself a highly competent Python programmer. Well done! These points are subtle, and your understanding of the language is becoming increasingly thorough as you continue here.

The tools from this lesson will allow you to use callables with greater flexibility and to better purpose. You've learned ways to write code that is able to collaborate with the interpreter and will allow you to accomplish many of your desired programming tasks more efficiently. Nice work!

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

Data Structures

Lesson Objectives

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

- organize data efficiently.
- create a two-dimensional array.

Organizing Data

In general, programming models the real world. Keep that in mind and it will help you to choose appropriate data representations for specific objects. This may *sound* pretty straightforward, but in fact, it takes a considerable amount of experience to get it right.

Initially, you might struggle to find the best data structure for an application, but ultimately working through those struggles will make you a better programmer. Of course you could bypass such challenges and follow some other programmer's prior direction, but I wouldn't recommend doing that. There's no substitute for working through programming challenges yourself. You develop a more thorough understanding of your programs when you make your own design decisions.

As you write more Python, you'll be able to accomodate increasingly complex data structures. So far, most of the structures we've created have been lists or dicts of the basic Python types—the immutables, like numbers and strings. However, there's no reason you can't use lists, tuples, dicts, or other complex objects (of your own creation or created using some existing library) as the elements of your data structures.

Data structures are important within your objects, as well. You define the behavior of a whole class of objects with a class statement. This class statement defines the behavior of each instance of the class by providing methods that the user can call to effect specific actions. Each instance has its own namespace though, which makes it appear like a data structure with behaviors common to all members of its class.

Handling Multi-Dimensional Arrays in Python

Python's "array" module provides a way to store a sequence of values of the same type in a compact representation that does not require Python object overhead for each value in the array. Array objects are one-dimensional, similar to Python lists, and most code actually creates arrays from an iterable containing the relevant values. With large numbers of elements, this can represent a substantial memory savings, but the features offered by this array type are limited. For full multi-dimensional arrays of complex data types, you would normally go to the (third-party, but open source) NumPy package. In most computer languages, multiple dimensions can be addressed by using multiple subscripts. So the Nth item in the Mth row of an array called D would be D(M, N) in Fortran (which uses parentheses for subscripting).

INTERACTIVE CONSOLE SESSION

```
>>> mylst = ["one", "two", "three"]
>>> mylst[1]
'two'
>>> mylst[1.3]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: list indices must be integers, not float
>>> mylst[(1, 3)]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: list indices must be integers, not tuple
>>>
```

A Python list may have only a single integer or a slice as an index; anything else will raise a TypeError exception such as "list indices must be integers."

A list is a one-dimensional array, with only a single length. A two-dimensional array has a size in each of two dimensions (often discussed as the numbers of rows and columns). Think of it as a sequence of one-

dimensional lists—an array of arrays. Similarly, consider a three-dimensional array as a sequence of two-dimensional arrays, and so on (although four-dimensional arrays are not used all that frequently).

In Python we can usually create a class to execute any task. You may remember that indexing is achieved by the use of the **__getitem__()** method. Let's create a basic class that reports the arguments that call that class's **__getitem__()** method. This will help us to see how Python indexing works.

The only two types that can be used as indexes on a sequence are *integers* and *slices*. The contents within the square brackets in the indexing construct may be more complex than a regular integer. You won't usually work directly with slices, because in Python you can get the same access to sequences using multiple subscripts, separated by colons (often referred to as *slicing notation*). You can *slice* a sequence with notation like **s[m:n]**, and you can even specify a third item by adding what is known as the *stride* (a stride of S causes only every Sth value to be included in the slice) using the form s[M:N:S]. Although there are no Python types that implement multi-dimensional arrays, the language is ready for them, and even allows multiple slices as subscripts. The Numpy package frequently incorporates slicing notation to help facilitate data subsetting.

INTERACTIVE CONSOLE SESSION

```
>>> class GI:
...     def __getitem__(self, *args, **kw):
...         print("Args:", args)
...         print("Kws: ", kw)
...
>>> gi = GI()
>>> gi[0]
Args: (0,)
Kws:  {}
>>> gi[0:1]
Args: (slice(0, 1, None),)
Kws:  {}
>>> gi[0:10:-2]
Args: (slice(0, 10, -2),)
Kws:  {}
>>> gi[1, 2, 3]
Args: ((1, 2, 3),)
Kws:  {}
>>> gi[1:2:3, 4:5:6]
Args: ((slice(1, 2, 3), slice(4, 5, 6)),)
Kws:  {}
>>> gi[1, 2:3, 4:5:6]
Args: ((1, slice(2, 3, None), slice(4, 5, 6)),)
Kws:  {}
>>> gi[(1, 2:3, 4:5:6)]
  File "<console>", line 1
    gi[(1, 2:3, 4:5:6)]
           ^
SyntaxError: invalid syntax
>>> (1, 2:3, 4:5:6)
  File "<console>", line 1
    (1, 2:3, 4:5:6)
        ^
SyntaxError: invalid syntax
>>>
```

Slices are allowed only as top-level elements of a tuple of subscripting expressions. Parenthesizing the tuple, or trying to use a similar expression outside of subscripting brackets, both result in syntax errors. A single integer index is passed through to the **__getitem__()** method without change. But the interpreter creates a special object called a *slice* object for constructs that contain colons. The slice object is passed through to the **__getitem__()** method. The last line in the example demonstrates that the interpreter allows us to use *multiple* slice notations as subscripts, and the **__getitem__()** method will receive a tuple of slice objects. This gives you the freedom to implement subscripting and slicing just about any way you want—of course, you have to understand how to use slice objects to take full advantage of the notation. For our purposes now, this isn't absolutely necessary, but the knowledge will be valuable later in many other contexts. The diagrams below summarize what we've learned so far about Python subscripting:

obj[M]

is equivalent to

obj.__getitem__(M)

Note	The above equivalence holds true whether M is an integer or a slice. In cases where the slice is provided as a single argument, it should be considered equivalent to one of the **__getitem__()** calls below.

obj[M:N]

is equivalent to

obj.__getitem__(slice(M, N, None))

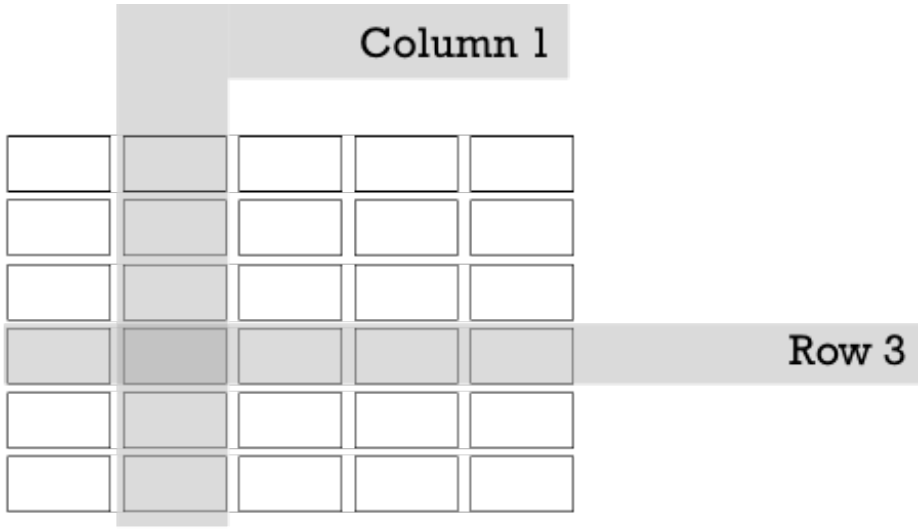
and

obj[M:N:S]

is equivalent to

obj.__getitem__(slice(M, N, S))

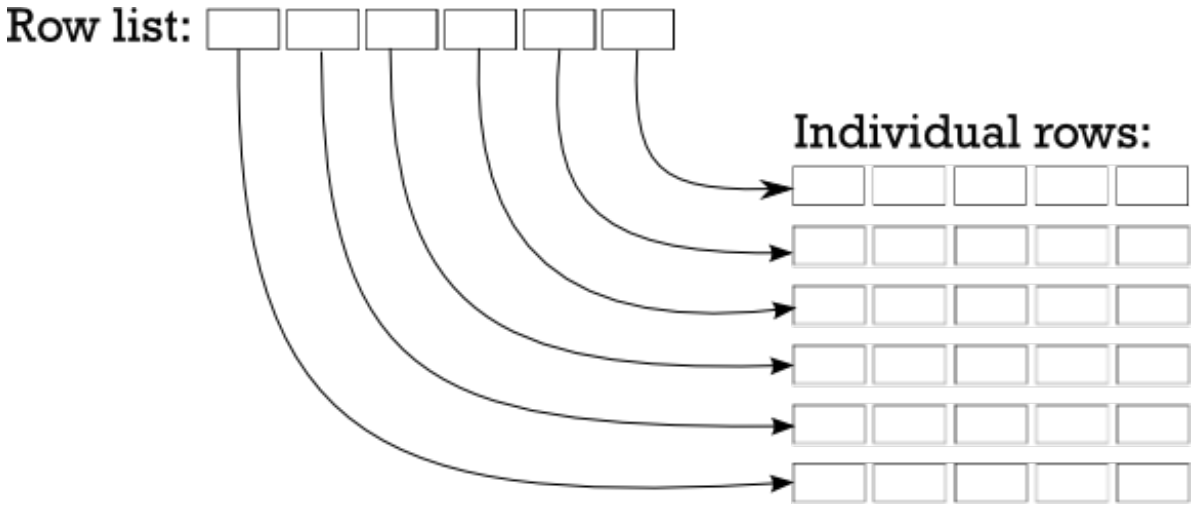
The list is a basic Python sequence, and like all the built-in sequence types, it is one-dimensional (that is, any item can be addressed with a single integer subscript of appropriate value). But multi-dimensional lists are often more convenient from a programmer's perspective, and, with the exception of the slicing notation, if you write a tuple of values as a subscript, then that tuple is passed directly through to the **__getitem__()** method. So it's possible to map tuples onto integer subscripts that can select a given item from an underlying list. Here's how a two-dimensional array should look to the programmer:



Representation of a 6x5 array
(numbering from zero)
Item [3, 1] highlighted

The most straightforward way to represent an array in Python is as a list of lists. Well actually, that would represent a *two*-dimensional array—a three- dimensional array would have to be a list of lists of lists, but you get the idea. So, in order to represent the array shown above, we could store it as either a list of rows or a list of columns. It doesn't really matter which type of list you choose, as long as you remain consistent. We'll use "row major order" (meaning we'll store a reference to the rows and then use the column number to index the element within that row) this time around.

For example, we could represent a 6x5 array as a six-element list, each item in that list consisting of a five-element list which represents a row of the array. To access a single item, you first have to index the row list with a row number (resulting in a reference to a row list), and then index *that* list to extract the element from the required column. Take a look:



Creating a Two-Dimensional Array

List of Lists Example

Let's write some code to create an *identity matrix*. This is a square array where every element is zero except for the main diagonal (the elements that have the same number for both row and column), and values of one. When you are dealing with complicated data structures, the print module often presents them more readably than a print.

While it might be easier to bang away at a console window for small pieces of code, it's good practice to define an API and write tests to exercise that API. This will allow you to try and test different representations efficiently, and you are able to improve your tests as you go. Create a **Python4_Lesson02** project, and in its

/src folder, create **testarray.py** as shown:

CODE TO TYPE: testarray.py

```
"""
Test list-of-list based array implementations.
"""
import unittest
import arr

class TestArray(unittest.TestCase):
    def test_zeroes(self):
        for N in range(4):
            a = arr.array(N, N)
            for i in range(N):
                for j in range(N):
                    self.assertEqual(a[i][j], 0)

    def test_identity(self):
        for N in range(4):
            a = arr.array(N, N)
            for i in range(N):
                a[i][i] = 1
            for i in range(N):
                for j in range(N):
                    self.assertEqual(a[i][j], i==j)

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

The tests are fairly limited at first, but even these basic tests allow you to detect gross errors in the code. Next, you'll need an **arr** module on which the test will operate. Let's start with a basic **arr module** for now. Create **arr.py** in the same folder as shown:

CODE TO TYPE: arr.py

```
"""
Naive implementation of list-of-lists creation.
"""

def array(M, N):
    "Create an M-element list of N-element row lists."
    rows = []
    for _ in range(M):
        cols = []
        for _ in range(N):
            cols.append(0)
        rows.append(cols)
    return rows
```

Run **testarray**; all tests pass.

OBSERVE:

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

OK
```

By now you may be able to devise ways to make the array code simpler. Right now, our code is straightforward, but rather verbose. Let's trim it down a little by using a list comprehension to create the individual rows. Modify your code as shown:

CODE TO EDIT: Modify arr.py

```
"""
Naive implementation of list-of-lists creation.
"""
def array(M, N):
    "Create an M-element list of N-element row lists."
    rows = []
    for _ in range(M):
        cols = []
        for _ in range(N):
            cols.append(0)
        rows.append(cols[0] * N)
    return rows
```

- All the tests still pass:

OBSERVE:

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

OK
```

At the moment, we are working strictly in two dimensions. But we are using "double subscripting"—**[M][N]**, rather than the "tuple of subscripts" notation—**[M, N]** that most programmers use (and that the Python interpreter is already prepared to accept). So let's modify our tests to use that notation, and verify that our existing implementation breaks when called without change. Modify **testarray.py** as shown:

CODE TO TYPE

```
"""
Test list-of-list array implementations using tuple subscripting.
"""
import unittest
import arr

class TestArray(unittest.TestCase):
    def test_zeroes(self):
        for N in range(4):
            a = arr.array(N, N)
            for i in range(N):
                for j in range(N):
                    self.assertEqual(a[i][j], 0)
                    self.assertEqual(a[i, j], 0)

    def test_identity(self):
        for N in range(4):
            a = arr.array(N, N)
            for i in range(N):
                a[i][i] = 1
                a[i, i] = 1
            for i in range(N):
                for j in range(N):
                    self.assertEqual(a[i][j], i==j)
                    self.assertEqual(a[i, j], i==j)

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

- The test output indicates that something isn't quite right in the array code after tuple-subscripting is used:

OBSERVE:

```
EE
======================================================================
ERROR: test_identity (__main__.TestArray)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python4_Lesson02\src\testarray.py", line 19, in test_identi
ty
    a[i, i] = 1
TypeError: list indices must be integers, not tuple

======================================================================
ERROR: test_zeroes (__main__.TestArray)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "V:\workspace\Python4_Lesson02\src\testarray.py", line 13, in test_zeroes
    self.assertEqual(a[i, j], 0)
TypeError: list indices must be integers, not tuple

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

FAILED (errors=2)
```

The only way to fix this is to define a class with a **__getitem__()** method, which will allow you direct access to the values passed as subscripts. This will make is easier to locate the correct element. Of course, the **__init__()** method has to create the lists and bind them to an instance variable that **__getitem__()** can access. The test code includes setting some array elements, so you also have to implement **__setitem__()**. (To respond properly to the del statement, a **__delitem__()** method should also be implemented, but this is not necessary for our immediate purposes.) Rewrite **arr.py** as shown:

CODE TO TYPE: arr.py

```
"""
Class-based list-of-lists allowing tuple subscripting.
"""

def array(M, N):
    "Create an M-element list of N-element row lists."
    rows = []
    for _ in range(M):
        rows.append([0] * N)
    return rows

class array:

    def __init__(self, M, N):
        "Create an M-element list of N-element row lists."
        self._rows = []
        for _ in range(M):
            self._rows.append([0] * N)

    def __getitem__(self, key):
        "Returns the appropriate element for a two-element subscript tuple."
        row, col = key
        return self._rows[row][col]

    def __setitem__(self, key, value):
        "Sets the appropriate element for a two-element subscript tuple."
        row, col = key
        self._rows[row][col] = value
```

Save it and rerun the test. With **__getitem__()** and **__setitem__()** in place on your array class, the tests pass again.

Using a Single List to Represent an Array

Using the standard subscripting API, you have built a way to reference two-dimensional arrays represented internally as a list of lists. If you wanted to represent a three-dimensional array, you'd have to change the code to operate on a list of lists of lists, and so on. However, the code might be more adaptable if it used just a single list and performed arithmetic on the subscripts to work out which element to access.

Now let's modify your current version of the arr module to demonstrate the principle on a 2-D array. We aren't going to extend the number of dimensions yet, but you might get an idea for how the code could be extended. Modify **arr.py** as shown:

CODE TO EDIT: arr.py

```
"""
Class-based single-list allowing tuple subscripting
"""

class array:

    def __init__(self, M, N):
        "Create an M element list of N element row lists."
        "Create a list long enough to hold M*N elements."
        self._rows = []
        for _ in range(M):
            self._rows.append([0] * N)
        self._data = [0] * M * N
        self._rows = M
        self._cols = N

    def __getitem__(self, key):
        "Returns the appropriate element for a two-element subscript tuple."
        row, col = key
        return self._rows[row][col]
        row, col = self._validate_key(key)
        return self._data[row*self._cols+col]

    def __setitem__(self, key, value):
        "Sets the appropriate element for a two-element subscript tuple."
        row, col = key
        self._rows[row][col] = value
        row, col = self._validate_key(key)
        self._data[row*self._cols+col] = value

    def _validate_key(self, key):
        """Validates a key against the array's shape, returning good tuples.
        Raises KeyError on problems."""
        row, col = key
        if (0 <= row < self._rows and
                0 <= col < self._cols):
            return key
        raise KeyError("Subscript out of range")
```

The changes that have been made here are pretty much invisible to the code that uses the module.

The **__init__()** method now initializes a single list that is big enough to hold all rows and columns. It also saves the array size in rows and columns. Previous versions could rely on access to the lists to detect any illegal values in the subscripts; now it has to be done explicitly because the location of the required element in the list now has to be calculated. We can no longer rely on IndexError exceptions to detect an out-of-bounds subscript. The current **__getitem__()** and **__setitem__()** methods use a **_validate_key()** method to verify that the subscript values do indeed fall within the required bounds before using them.

Although all existing tests pass, this detail about the index bounds checking reminds us to add tests to verify that the logic works and that a KeyError exception is raised when illegal values are used. The resulting changes are not complex. Modify **testarry.py** as shown:

CODE TO EDIT: testarray.py

```
"""
Test list-of-list array implementations using tuple subscripting.
"""
import unittest
import arr

class TestArray(unittest.TestCase):
    def test_zeroes(self):
        for N in range(4):
            a = arr.array(N, N)
            for i in range(N):
                for j in range(N):
                    self.assertEqual(a[i, j], 0)

    def test_identity(self):
        for N in range(4):
            a = arr.array(N, N)
            for i in range(N):
                a[i, i] = 1
            for i in range(N):
                for j in range(N):
                    self.assertEqual(a[i, j], i==j)

    def _index(self, a, r, c):
        return a[r, c]

    def test_key_validity(self):
        a = arr.array(10, 10)
        self.assertRaises(KeyError, self._index, a ,-1, 1)
        self.assertRaises(KeyError, self._index, a ,10, 1)
        self.assertRaises(KeyError, self._index, a ,1, -1)
        self.assertRaises(KeyError, self._index, a ,1, 10)

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

When all three tests pass, you can be confident in your bounds-checking logic. Keep in mind that it's just as important to make sure your program fails when it should, as it is to make sure it runs correctly when it should!

OBSERVE

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

OK
```

As long as the API remains the same, you'll have considerable flexibility and programming technique options. Let's consider alternative representations.

Using an array.array instead of a List

The array module defines a single data type (also called "array"), which is similar to a list, except that it stores homogeneous values (each cell can hold values of a given type only, that type being passed when the array is created). The changes required to use such an array instead of a list are minimal. Modify **arr.py** as shown:

CODE TO EDIT: arr.py

```
"""
Class-based array allowing tuple subscripting
"""
import array as sys_array

class array:

    def __init__(self, M, N):
        "Create a list long enough to hold M*N elements."
        "Create an M-element list of N-element row lists."
        self._data = [0] * M * Nsys_array.array("i", [0] * M * N)
        self._rows = M
        self._cols = N

    def __getitem__(self, key):
        "Returns the appropriate element for a two-element subscript tuple."
        row, col = self._validate_key(key)
        return self._data[row*self._cols+col]

    def __setitem__(self, key, value):
        "Sets the appropriate element for a two-element subscript tuple."
        row, col = self._validate_key(key)
        self._data[row*self._cols+col] = value

    def _validate_key(self, key):
        """Validates a key against the array's shape, returning good tuples.
        Raises KeyError on problems."""
        row, col = key
        if (0 <= row < self._rows and
                0 <= col < self._cols):
            return key
        raise KeyError("Subscript out of range")
```

The testing doesn't change in this case (note that the updated code in the arr module requires the numbers stored in the array.array to be integers), and so your tests pass immediately. The advantage of this implementation (for applications using integer data) is most evident when you're working with extremely large data structures. In these cases, values can be packed closely together within memory, because the array.array structure does not store them as Python values. This could save large amounts of memory overhead with large datasets, and further smaller savings would result from not having to allocate memory for the lists that refer to rows or individual values.

Using a dict instead of a List

Some mathematical techniques use "sparse" data sets. These are usually representations of very large data sets where the majority of the values are zero (and therefore do not need to be duplicated). This technique lends itself to using a dict to store the non-zero values using the subscript tuple passed in to the **__getitem__()** method.

Since the data storage element does not provide any bounds checking, the methods should still do that. There is no need to initialize the dict with zeroes, because the absence of a value implies a zero! Modify **arr.py** as shown:

CODE TO EDIT: arr.py

```
"""
Class-based dict allowing tuple subscripting and sparse data
"""
import array as sys_array

class array:

    def __init__(self, M, N):
        "Create an M-element list of N-element row lists."
        self._data = sys_array.array("i", [0] * M * N)
        self._data = {}
        self._rows = M
        self._cols = N

    def __getitem__(self, key):
        "Returns the appropriate element for a two-element subscript tuple."
        row, col = self._validate_key(key)
        try:
            return self._data[row, col]
        except KeyError:
            return  0
        return self._data[row*self._cols+col]

    def __setitem__(self, key, value):
        "Sets the appropriate element for a two-element subscript tuple."
        row, col = self._validate_key(key)
        self._data[row*self._cols+col] = value
        self._data[row, col] = value

    def _validate_key(self, key):
        """Validates a key against the array's shape, returning good tuples.
        Raises KeyError on problems."""
        row, col = key
        if (0 <= row < self._rows and
                0 <= col < self._cols):
            return key
        raise KeyError("Subscript out of range")
```

- Save it and run the test again. The testing is somewhat simplified in this version, since zero values do not need to be asserted. (Note that the current **__setitem__()** method is deficient in some ways; the storage of a zero should result in the given key being removed from the dict if present).

Summary

So now we have loads of options at our disposal to complete our various Python tasks. Having so much flexibility enables you to choose specific techniques to suit your specific needs. With some practice, you'll be able to make the most efficient compromises between efficient use of storage and adequate computation speed. You're doing a fine job so far! See you in the next lesson...

When you finish the lesson, don't forget to return to the syllabus and complete the homework.

Delegation and Composition

Lesson Objectives

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

- extend functionality by inheritance.
- execute more complex delegation.
- extend functionality by composition.
- utilize recursive composition.

Let's get right to it then, shall we?

In Python, it's unusual to come across deep inheritance trees (E inherits from D which inherits from C which inherits from B which inherits from A). While such program structures are possible, they can become unwieldy quickly. If you want to implement a dict-like object with some additional properties, you could choose to **inherit** from dict and extend the behavior, or you could decide to **compose** your own object from scratch and make use of a dict internally to provide the desired dict-like properties.

Extending Functionality by Inheritance

Suppose you want to make your program keep count of how many items have been added (that is, how many times a previously non-existent key was bound in the table. If the key already exists, it isn't an addition—it's a replacement). With inheritance, you'd do it like this:

INTERACTIVE CONSOLE SESSION

```
>>> class Dict(dict):
...     def __init__(self, *args, **kw):
...         dict.__init__(self, *args, **kw)
...         self.adds = 0
...     def __setitem__(self, key, value):
...         if key not in self:
...             self.adds += 1
...         dict.__setitem__(self, key, value)
...
>>> d = Dict(a=1, b=2)
>>> print("Adds:", d.adds)
Adds: 0
>>> d["newkey"] = "add"
>>> print("Adds:", d.adds)
Adds: 1
>>> d["newkey"] = "replace"
>>> print("Adds:", d.adds)
Adds: 1
>>>
```

This code behaves as we'd expect. Albeit limited, it provides functionality over and above that of dict objects.

OBSERVE:

```
class Dict(dict):
    def __init__(self, *args, **kw):
        self.adds = 0
        dict.__init__(self, *args, **kw)
    def __setitem__(self, key, value):
        if key not in self:
            self.adds += 1
        dict.__setitem__(self, key, value)
```

Our Dict class inherits from the dict built-in. Because this Dict class needs to perform some initialization, it has to make sure that the dict object initializes properly. The dict accomplishes this with an explicit call to the parent object (dict) with the arguments that were provided to the initializing call to the class. **dict.__init__(self, *args, **kw)** passes all the positional and keyword arguments that the caller passes, beginning with providing the current instance as an explicit first argument (remember, the automatic provision of the instance argument only happens when a method is called on an *instance*—this method is being called on the *superclass*).

Because the dict type can be called with many different arguments, it is necessary to adopt this style, so that this dict can be used just like a regular dict. We might say that the Dict object *delegates* most of its initialization to its superclass. Similarly, the only difference between **the __setitem__()** method and a pure dict appears when testing to determine **whether the key already exists in the dict**, and if not, incrementing the "add" count. The remainder of the method is implemented by calling dict's superclass (the standard dict) to perform the normal item assignment, by calling its **__setitem__()** method with the same arguments: **dict.__setitem__(self, key, value)**.

The initializer function does not call the **__setitem__ ()** method to add any initial elements—the adds attribute still has the value zero immediately after creation, despite the fact that the instance was created with two items.

Note We didn't do it here, but if you are going to deliver code to paying customers, or if you expect the code to see heavy use, you'll want to run tests that verify it operates correctly. Writing tests can be difficult, but when something is going into production, it's important to have a bank of tests available. That way, if anyone refactors your code, they can do so with some confidence that if the tests still pass, they haven't broken anything.

The Dict class inherits from dict. This is appropriate because most of the behavior you want is standard dict behavior. Since both the **__init__()** and **__setitem__()** methods of Dict call the equivalent methods of dict as a part of their code, we say that those methods *extend* the corresponding dict methods.

More Complex Delegation

In general, the more of a particular object's behaviors you need, the more likely you are to inherit from it. But if only a small part of the behavior you require is provided by an existing class, you might choose to create an instance of that class and bind it to an instance variable of your own class instead. The approach is similar, but does not use inheritance. Let's take a look at that:

INTERACTIVE CONSOLE SESSION

```
>>> class MyDict:
...     def __init__(self, *args, **kwargs):
...         self._d = dict(*args, **kwargs)
...     def __setitem__(self, key, value):
...         return self._d.__setitem__(key, value)
...     def __getitem__(self, key):
...         return self._d.__getitem__(key)
...     def __delitem__(self, key):
...         return self._d.__delitem__(key)
...
>>> dd = MyDict(wynken=1, blynken=2)
>>> dd['blynken']
2
>>> dd['nod'] -->
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __getitem__
KeyError: 'nod'
>>> dd['nod'] = 3
>>> dd['nod']
3
>>> del dd['nod']
>>> dd.keys()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyDict' object has no attribute 'keys'
>>>
```

Here the **MyDict** class creates a dict in its **__init__()** method and binds it to the instance's **_d** variable. Three methods of the MyDict class are delegated to that instance, but none of the other methods of the dict are available to the MyDict user (which may or may not be what you intend). In this particular case, the MyDict class doesn't subclass dict, and so not all dict methods are available.

The final attempt to access the keys of the MyDict instance shows one potential shortcoming of this approach: methods of the underlying object have to be made available explicitly. This technique can be useful when only a limited subset of behaviors is required, along with other functionality (provided by additional methods) not available from the base type. Where most of the behaviors of the base type are required, it is usually better to use inheritance, and then override the methods that you don't want to make available with a method that raises an exception.

Extending Functionality by Composition

Object composition allows you to create complex objects by using other objects, typically bound to instance variables. An example where you might use such a complex object is during an attempt to simulate Python's namespace access. You have already seen that Python gives many objects a namespace, and you know that the interpreter, when looking for an attribute of a particular name, will first look in the instance's namespace, next in the instance's class's namespace, and so on until it gets to the "top" of the inheritance chain (which is the built-in object class).

It is relatively straightforward to model a Python namespace; they are almost indistinguishable from dicts. Names are used as keys, and the values associated with the names are the natural parallel to the values of the variables with those names. Multiple dicts can be stored in a list, with the dict to be searched placed first, as the lowest-numbered element.

INTERACTIVE CONSOLE SESSION

```
>>> class Ns:
...     def __init__(self, *args):
...         "Initialize a tuple of namespaces presented as dicts."
...         self._dlist = args
...     def __getitem__(self, key):
...         for d in self._dlist:
...             try:
...                 return d[key]
...             except KeyError:
...                 pass
...         raise KeyError("{!r} not present in Ns object".format(key))
...
>>> ns = Ns(
...         {"one": 1, "two": 2},
...         {"one": 13, "three": 3},
...         {"one": 14, "four": 4}
...     )
>>>
>>> ns["one"]
1
>>> ns["four"]
4
>>>
```

The Ns class uses a list of dicts as its primary data store, and doesn't call any of their methods directly. It does call their methods indirectly though, because the **__getitem__()** method iterates over the list and then tries to access the required element from each dict in turn. Each failure raises a KeyError exception, which is ignored by the pass statement to move on to the next iteration. So, effectively the **__getitem__()** method searches a list of dicts, stopping as soon as it finds something to return. That is why **ns["one"]** returned 1. While 14 is associated with the same key, this association takes place in a dict later in the list and so is never considered; the function has already found the same key in an earlier list and returned with that key's value.

Think of an Ns object as being "composed" of a list and dicts. Technically, any object can be considered as being composed of all of its instance variables, but we don't normally regard composition as extending to simple types such as numbers and strings. If you think about Python namespaces they act a bit like this: there are often a number of namespaces that the interpreter needs to search. Adding a new namespace (like a new layer of inheritance does to a class's instances, for example) would be the equivalent on inserting a new dict at position 0 (Do you know which list method will do that?).

Recursive Composition

Some data structures are simple, others are complex. Certain complex data structures are composed of other instances of the same type of object; such structures are sometimes said to be *recursively composed*. A typical example is the tree, used in many languages to store data in such a way that it can easily be retrieved both randomly and sequentially (in the order of the keys). The tree is made up of nodes. Each node contains data and two pointers. One of the data elements will typically be used as the *key*, which determines the ordering to be maintained among the nodes. The first pointer points to a subtree containing only nodes with key values that are less than the key value of the current node, and the second points to a subtree containing only nodes with key values that are greater than that of the current node.

Either of the subtrees may be empty (there may not *be* any nodes with the required key values); if both subtrees are empty, the node is said to be a *leaf node*, containing only data. If the relevant subtree is empty, the corresponding pointer element will have the value None (all nodes start out containing only data, with None as the left and right pointers).

Note In a real program, the nodes would have other data attached to them as well as the keys, but we have omitted this feature to allow you to focus on the necessary logic to maintain a tree.

Create a new PyDev project named **Python4_Lesson03** and assign it to the **Python4_Lessons** working set. Then, in your **Python4_Lesson03/src** folder, create **mytree.py** as shown:

CODE TO TYPE:

```
'''
Created on Aug 18, 2011

@author: sholden
'''
class Tree:
    def __init__(self, key):
        "Create a new Tree object with empty L & R subtrees."
        self.key = key
        self.left = self.right = None
    def insert(self, key):
        "Insert a new element into the tree in the correct position."
        if key < self.key:
            if self.left:
                self.left.insert(key)
            else:
                self.left = Tree(key)
        elif key > self.key:
            if self.right:
                self.right.insert(key)
            else:
                self.right = Tree(key)
        else:
            raise ValueError("Attempt to insert duplicate value")
    def walk(self):
        "Generate the keys from the tree in sorted order."
        if self.left:
            for n in self.left.walk():
                yield n
        yield self.key
        if self.right:
            for n in self.right.walk():
                yield n

if __name__ == '__main__':
    t = Tree("D")
    for c in "BJQKFAC":
        t.insert(c)

    print(list(t.walk()))
```

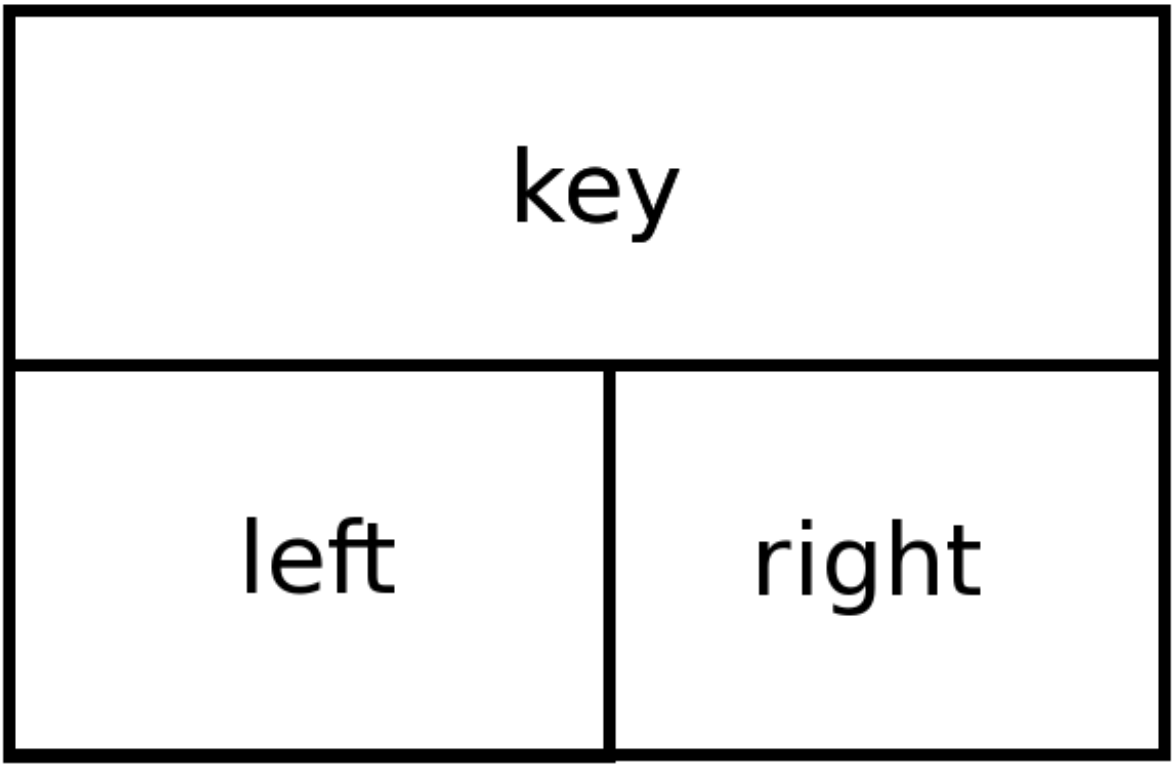
Here again we chose not to have you write tests for your code, but we do test it rather informally with the code following the class declaration. The tree as created, consists of a single node. After creation, a loop inserts a number of characters, and then finally the **walk()** method is used to visit each node and print out the value of each data element.

The root of the tree is a Tree object, which in turn may point to other Tree nodes. This means that each subtree has the same structure as its parent, which implies that the same methods/algorithms can be used on the subtrees. This can make the processing logic for recursive structures quite compact.

The **insert()** method locates the correct place for the insertion by comparing the node key with the key to be inserted. If the new key is less than the node's key, it must be positioned in the left subtree, if greater, in the right subtree. If there isn't a subtree there (indicated by the left or right attribute having a value of None), the newly-created node is added as its value. If the subtree exists, *its* insert method is called to place it correctly. So not only is the data structure recursive, so is the algorithm to deal with it!

The **walk()** method is designed to produce values from the nodes in sorted order. Again the algorithm is recursive: first it walks the left subtree (if one exists), then it produces the current node (it yields the key value, but clearly the data would be preferable, either instead of or in addition to the key value, if it were being stored—here we are more concerned with the basics of the tree structure than with having the tree carry data, which could easily be added as a new Tree instance variable passed in to the **__init__()** call on creation).

In essence, a Tree is a "root node" (the first one added, in this case with key "D") that contains a key value and two subtrees—the first one for key values less than that of the root node, the second for key values greater than that of the root node. The subtrees, of course, are defined in exactly the same way, and so can be processed in the same way. Recursive data structures and recursive algorithms tend to go together. The Tree offers a fairly decent visual representation for your brain to latch onto:

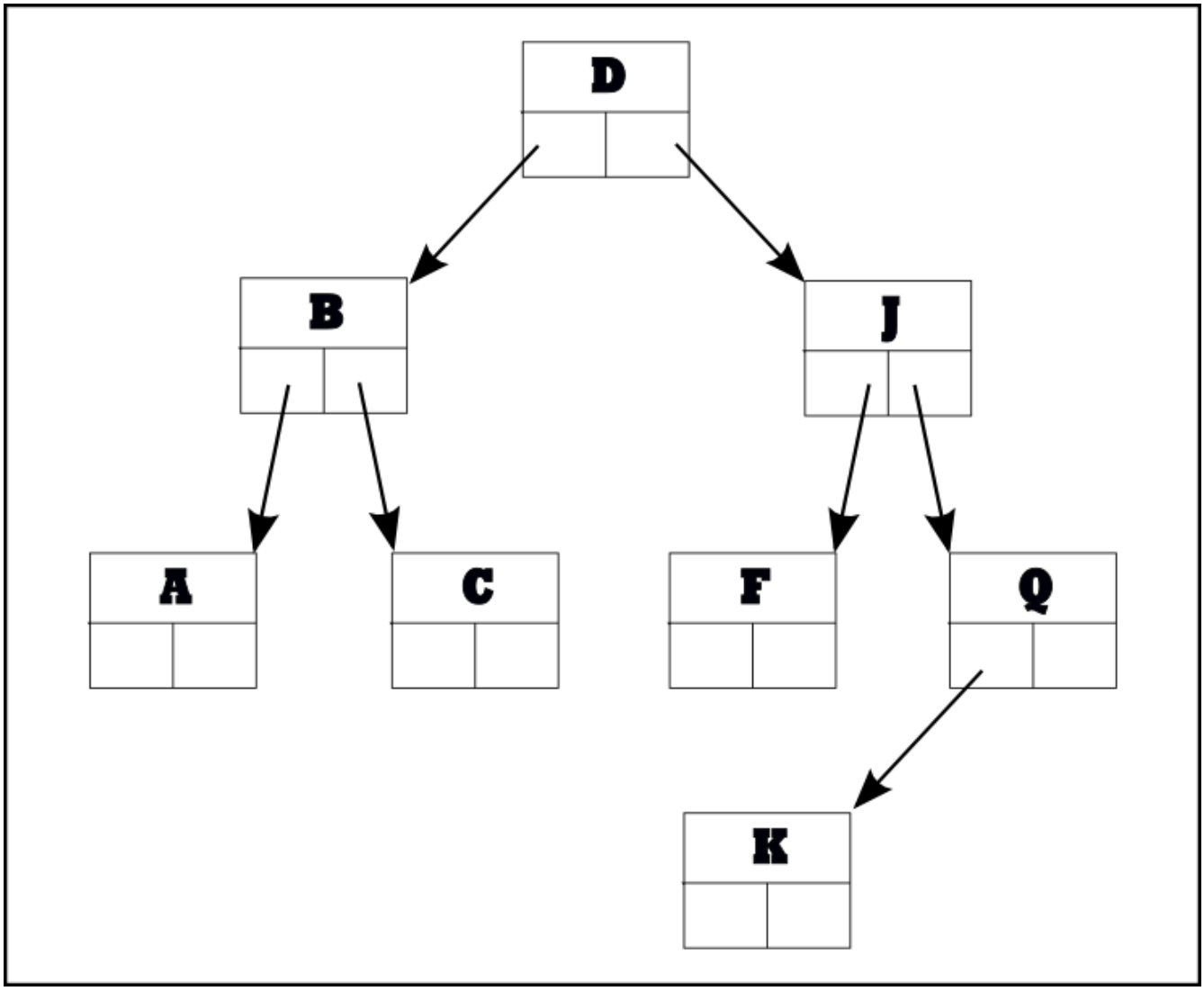


Such recursive algorithms aren't quite the same as delegation, but still, you could think of walk() and insert() as delegating a part of the processing to the subtrees. When you run **tree.py**, you'll see this:

OBSERVE:

```
['A', 'B', 'C', 'D', 'F', 'J', 'K', 'Q']
```

This is how the tree actually stores elements in terms of Tree objects referencing each other (the diagonal lines represent Python references, the letters are the keys):



Although the keys were added in random order, the walk() method prints them in the correct order because it prints out the keys of the left subtree followed by the key of the root node, followed by the keys of the right subtree (it deals with subtrees in the same way).

Great work! You've actually used composition in examples and projects. Now that you have a handle on composition, ponder the many ways you could incorporate it into other programs!

When you finish the lesson, don't forget to return to the syllabus and complete the homework.

Publish and Subscribe

Lesson Objectives

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

- structure programs.
- publish objects.
- validate requests and identify output.
- subscribe to objects.

In this lesson, we'll go over program structuring, as well as **Publish** and **Subscribe**.

On Program Structure

Ideally, every part of your program will communicate via known APIs only, but accomplishing that can be a real challenge. When you are writing frameworks to be used in a wide variety of circumstances, it can be difficult to predict what the environment will look like. Data must be produced, but it may be consumed by a variety of functions. Consider a spreadsheet, for example. It may display both a bar chart and a pie chart of the same data. How does the code that updates the cells as users type in new numbers know to update the graphics, and how many graphics there are? The answer lies in a generic technique known as "publish-and-subscribe", which is a general mechanism to allow flexible distribution of data.

Publish and Subscribe

Thanks to publish-and-subscribe and similar systems, data producers do not need to know in advance who will be using their data. The term "data producer" is deliberately vague, because publish-and-subscribe is a broad and encompassing architectural pattern. A data producer (the "publisher" element in publish-and-subscribe) might be a stock price ticker that periodically spits out new prices for stocks, or a weather forecasting program that produces new forecasts every six hours, or even the lowly ticket machine that provides people with numbers to take turns at a grocery counter. Anyone who wants to make use of the data must subscribe (typically by calling a method of the producer object to "register" a subscription) and then when new data is available, it is distributed to all subscribers by the publisher calling a method of each of the subscribed objects with the new information as an argument.

This "loosens the coupling" between the producers and consumers of data, allowing each to be written in a general way, pretty much independent of each other. Each subscriber needs to know only about its own relationship with the publisher, regardless of any other subscriber.

Publish and Subscribe in Action

Suppose you have a class Publisher, whose instances can be given objects to publish, and that a number of consumers are potentially interested in consuming that "data feed." The Publisher class will need methods to allow the subscribers to subscribe when they want to start receiving the feed and unsubscribe when they no longer require it.

The consumers, in turn, have to know how the Publisher will transmit the data to them, which will normally be achieved by calling one of its methods. So consumers may need to provide an API to satisfy the requirements of the Publisher. We'll create an example.

For our purposes, we'll write a module that asks for lines of input from the user, and then distributes the lines to any subscribed consumers. The subscriber interface will have subscribe and unsubscribe methods that add and remove items from the publisher's subscriber list. Subscribers must provide a "process" method, which the publisher will call with each new input.

We will have the subscribers print the input string after processing it in basic, but distinguishable ways. In the first example, subscribers print out the uppercase version of the string they've received.

Create a **Python4_Lesson04** project and add it to your **Python4_Lessons** working set. Then, create **pubandsub.py** in your **Python4_Lesson04/src** folder as shown:

CODE TO TYPE:

```
class Publisher:
    def __init__(self):
        self.subscribers  = []
    def subscribe(self, subscriber):
        self.subscribers.append(subscriber)
    def unsubscribe(self, subscriber):
        self.subscribers.remove(subscriber)
    def publish(self, s):
        for subscriber in self.subscribers:
            subscriber.process(s)

if __name__ == '__main__':
    class SimpleSubscriber:
        def __init__(self, publisher):
            publisher.subscribe(self)
            self.publisher = publisher
        def process(self, s):
            print(s.upper())

    publisher = Publisher()
    for i in range(3):
        newsub = SimpleSubscriber(publisher)
        line = input("Input {}: ".format(i))
        publisher.publish(line)
```

The program asks you for three lines of input. The first is echoed in uppercase once, the second twice, and the third three times, because each time through the loop, a new subscriber is subscribed to the publisher.

OBSERVE:

```
Input 0: pub
PUB
Input 1: and
AND
AND
Input 2: sub
SUB
SUB
SUB
```

The Publisher keeps a list of subscribers (which starts out empty). Subscribing an object appends it to the subscriber list; unsubscribing an object removes it. The SimpleSubscriber object takes a publisher as an argument to the **__init__()** method and immediately subscribes to the publisher.

These same principles can be applied to programs you may already use. For example, a spreadsheet program may have to process spreadsheets where there are multiple graphics based on the data, all of which must be updated as the data changes. One way to arrange that is to enlist the graphics as subscribers to an event stream publisher, which publishes an alert every time any change is made to the data. To avoid unnecessary computing, the event stream publisher might publish the event after a change only when no further changes were made to the data within a fixed (and preferably short) period of time.

We can refine this process further in various ways because it allows very *loose coupling* between the publisher and the subscriber: neither needs to have advance knowledge of the other, and the connections are created at run-time rather than determined in advance. We like loose coupling in systems design because it's flexible and allows dynamic relationships between objects.

Validating Requests and Identifying Output

Our initial implementation is defective in a couple of ways. First, there is nothing to stop a given subscriber from being subscribed multiple times. Similarly, there is nothing present to check whether a subscriber requesting unsubscription (code not yet exercised in the main program) is actually subscribed. Passing a nonexistent subscriber would cause the list's remove() method to raise an exception:

OBSERVE:

```
>>> [1, 2, 3].remove(4)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

In order to make the message associated with the exception easier to understand, you'll want to trap it or test beforehand for the condition that would cause the exception and then raise your own, more meaningful, exception.

Finally, the original version of our program does not identify which specific subscriber is responsible for an individual message. We want it to identify the culprit though, because that will make the operation of the program easier to understand. Let's revise it so that each subscriber instance takes an additional argument (its name), which it will then use to identify all of its output. Modify **pubandsub.py** to check for errors and identify subscribers

CODE TO TYPE:

```
class Publisher:
    def __init__(self):
        self.subscribers  = []
    def subscribe(self, subscriber):
        if subscriber in self.subscribers:
            raise ValueError("Multiple subscriptions are not allowed")
        self.subscribers.append(subscriber)
    def unsubscribe(self, subscriber):
        if subscriber not in self.subscribers:
            raise ValueError("Can only unsubscribe subscribers")
        self.subscribers.remove(subscriber)
    def publish(self, s):
        for subscriber in self.subscribers:
            subscriber.process(s)

if __name__ == '__main__':
    class SimpleSubscriber:
        def __init__(self, name, publisher):
            publisher.subscribe(self)
            self.name = name
            self.publisher = publisher
        def process(self, s):
            print(self.name, ":", s.upper())

    publisher = Publisher()
    for i in range(3):
        newsub = SimpleSubscriber("Sub"+str(i), publisher)
        line = input("Input {}: ".format(i))
        publisher.publish(line)
```

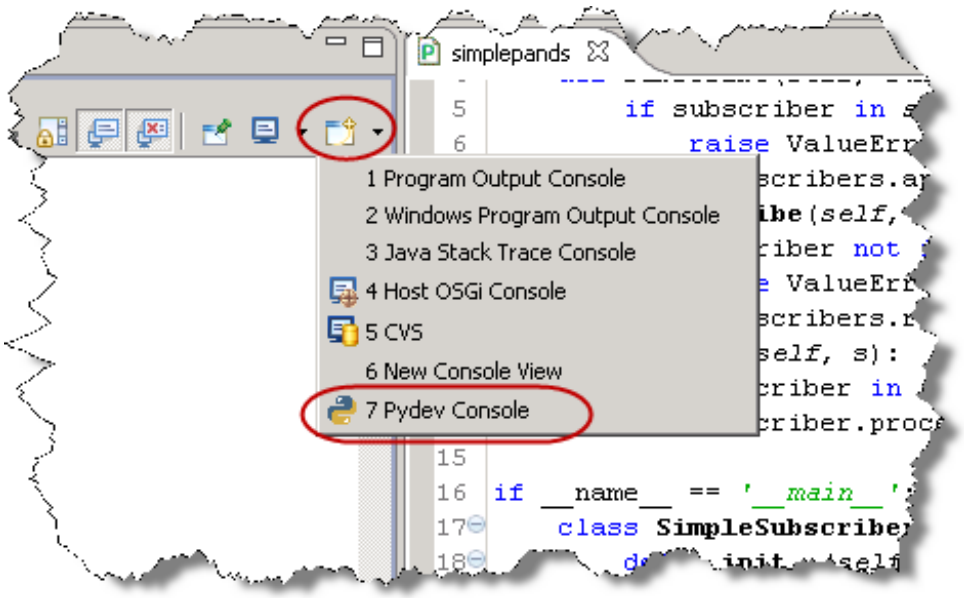
This version of the program doesn't actually trigger any of the newly-added exceptions, but the inclusion of the tests makes our code more robust. The SimpleSubscriber.process() method identifies each output line with the name of the instance that was responsible for it, which can be especially helpful in more complex situations. The code that creates the subscribers generates names such as "Sub0", "Sub1" and so on for the subscribers. You should see output that looks like this:

OBSERVE:

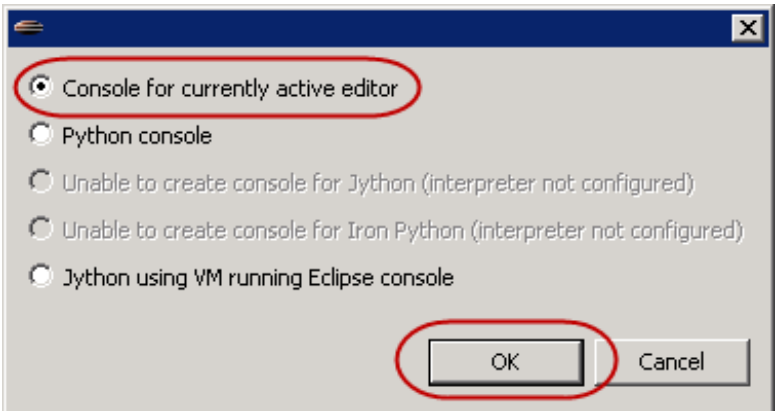
```
Input 0: sub
Sub0 : SUB
Input 1: and
Sub0 : AND
Sub1 : AND
Input 2: pub
Sub0 : PUB
Sub1 : PUB
Sub2 : PUB
```

If we were to write unit tests for this code, we might include an assertRaises() test to ensure that the double-subscription and attempts to remove non-subscribed objects were handled correctly. In the absence of unit tests, we should at least make sure that exceptions will be raised under expected circumstances. We can do that in an interactive console with the help of Eclipse.

First, make sure that you activate the editor session containing the **pubandsub.py** source.Then, in the Console pane, click **Open Console** and select **PyDev Console** from the drop-down menu that appears:



You will see a dialog asking you which type of console window you want to create. Select **Console for currently active editor** and click **OK**:



Now you will be able to import modules from the Python4_Lesson04/src directory. Next, verify that exceptions are properly raised:

INTERACTIVE CONSOLE SESSION

```
>>> from pubandsub import Publisher
>>> publisher = Publisher()
>>> publisher.unsubscribe(None)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "pubandsub.py", line 16, in unsubscribe
    raise ValueError("Can only unsubscribe subscribers")
ValueError: Can only unsubscribe subscribers
>>> publisher.subscribe(None)
>>> publisher.subscribe(None)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "pubandsub.py", line 12, in subscribe
    raise ValueError("Multiple subscriptions are not allowed")
ValueError: Multiple subscriptions are not allowed
>>>
```

Since exceptions appear to be raised under the correct circumstances, we could proceed without modifying the code further, but it's a good idea to copy and paste the interactive session into your source as a doctest. A simple copy-and-paste from the console panel is not adequate, however, because the console is designed to let you copy and paste *only* the code, so when you copy from the interactive session in Eclipse, the necessary prompt strings (">>> " and "... ") are absent from the pasted content. doctest and Eclipse don't always play nicely together. It's a good thing Eclipse has so many other useful features.

So far our program has not tested the non-error branch of the unsubscribe code. We'll perform that test next by restricting the number of subscribers. This can be done either internally (from within the Publisher.subscribe() method, for example) or by truncating the subscription list from the main loop. We're going to do the latter. We'll add a few loops to make sure that the strategy is properly tested. After each new subscription, we'll remove the least recent if the length of the subscription list exceeds three. This will ensure that no input sees more than three responses. Modify **pubandsub.py** as shown below

CODE TO TYPE:

```
class Publisher:
    def __init__(self):
        self.subscribers  = []
    def subscribe(self, subscriber):
        if subscriber in self.subscribers:
            raise ValueError("Multiple subscriptions are not allowed")
        self.subscribers.append(subscriber)
    def unsubscribe(self, subscriber):
        if subscriber not in self.subscribers:
            raise ValueError("Can only unsubscribe subscribers")
        self.subscribers.remove(subscriber)
    def publish(self, s):
        for subscriber in self.subscribers:
            subscriber.process(s)

if __name__ == '__main__':
    class SimpleSubscriber:
        def __init__(self, name, publisher):
            publisher.subscribe(self)
            self.name = name
            self.publisher = publisher
        def process(self, s):
            print(self.name, ":", s.upper())

    publisher = Publisher()
    for i in range(5):
        newsub = SimpleSubscriber("Sub"+str(i), publisher)
        if len(publisher.subscribers) > 3:
            publisher.unsubscribe(publisher.subscribers[0])
        line = input("Input {}: ".format(i))
        publisher.publish(line)
        line = input("Input {}: ".format(i))
        publisher.publish(line)
```

This code is not much different from the last example, except that there are never more than three responses to any input, which indicates that the unsubscribe function is working correctly. Each time the subscriber count exceeds three it is trimmed from the left:

OBSERVE:

```
Input 0: sub
Sub0 : SUB
Input 1: and
Sub0 : AND
Sub1 : AND
Input 2: pub
Sub0 : PUB
Sub1 : PUB
Sub2 : PUB
Input 3: more
Sub1 : MORE
Sub2 : MORE
Sub3 : MORE
Input 4: inputs
Sub2 : INPUTS
Sub3 : INPUTS
Sub4 : INPUTS
```

Making the Algorithm More General

At present, the publisher requires subscribers to have a "process" method, which it calls to have each subscriber process the published data. This works well enough, but it does constrain the nature of the subscribers. For example, there is no way to subscribe functions, because there is no way to add a method to a function.

Let's modify the program so that it registers the callable method directly instead of registering an instance and then calling a specific method. Our program will then allow any callable to be registered. We'll verify this by defining a simple function and registering it with the publisher before the loop begins. Modify **pubandsub.py** to allow registration of any callable:

CODE TO TYPE:

```
class Publisher:
    def __init__(self):
        self.subscribers  = []
    def subscribe(self, subscriber):
        if subscriber in self.subscribers:
            raise ValueError("Multiple subscriptions are not allowed")
        self.subscribers.append(subscriber)
    def unsubscribe(self, subscriber):
        if subscriber not in self.subscribers:
            raise ValueError("Can only unsubscribe subscribers")
        self.subscribers.remove(subscriber)
    def publish(self, s):
        for subscriber in self.subscribers:
            subscriber.process(s)

if __name__ == '__main__':
    def multiplier(s):
        print(2*s)

    class SimpleSubscriber:
        def __init__(self, name, publisher):
            publisher.subscribe(self)
            self.name = name
            self.publisher = publisher
            publisher.subscribe(self.process)
        def process(self, s):
            print(self, ":", s.upper())
        def __repr__(self):
            return self.name

    publisher = Publisher()
    publisher.subscribe(multiplier)
    for i in range(6):
        newsub = SimpleSubscriber("Sub"+str(i), publisher)
        line = input("Input {}: ".format(i))
        publisher.publish(line)
        if len(publisher.subscribers) > 3:
            publisher.unsubscribe(publisher.subscribers[0])
```

The SimpleSubscriber object now registers its (bound) process method as a callable, and the Publisher.publish() method calls the subscribers directly rather than calling a method of the subscriber. This makes it possible to subscribe functions to the Publisher:

OBSERVE:

```
Input 0: pub
pubpub
Sub0 : PUB
Input 1: and
andand
Sub0 : AND
Sub1 : AND
Input 2: sub
subsub
Sub0 : SUB
Sub1 : SUB
Sub2 : SUB
Input 3: and
Sub0 : AND
Sub1 : AND
Sub2 : AND
Sub3 : AND
Input 4: dub
Sub1 : DUB
Sub2 : DUB
Sub3 : DUB
Sub4 : DUB
Input 5: and
Sub2 : AND
Sub3 : AND
Sub4 : AND
Sub5 : AND
```

Note The full "publish and subscribe" algorithm is general enough to allow communication between completely different processes. Technically, we have been studying a subset of publish-and-subscribe also referred to as "the observer pattern."

A Note on Debugging

Eclipse has some advanced debugging features, but we've ignored them. You won't always have Eclipse at your disposal (at least when you aren't in the lab), so instead, we've directed our attention to assuring your code through testing.

The relatively simple expedient of *inserting print()* calls in your code is good enough to solve many problems, and in the upcoming project the most important part of the exercise is to use this technique to discover exactly how the suggested modification breaks the program. See you in the next lesson!

When you finish the lesson, don't forget to return to the syllabus and complete the homework.

Optimizing Your Code

Lesson Objectives

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

- focus your attention to the proper elements from the beginning.
- use the profile module.
- identify which elements should be optimized.
- optimize.

Start with Correctness

"Speed is fine, but accuracy is everything."
-Wyatt Earp

Inexperienced programmers often devote the majority of their attention to speed and performance. This is a common mistake that can often lead to additinal mistakes made as a result of working with accelerated program speeds too early on in the programming process. During development, your initial focus should be on producing programs that work correctly and are supported by tests. When you do begin to consider speed and performance, you're likely to alter your code; that's when tests will be indispensible. If your changes break your tests, you'll need to fix your code before you address issues of speed and performance. The prevailing programmer's wisdom applies, "First, make it work, then make it faster."

When you write a working program, it's generally fast enough already. That isn't to say that your programs can't be made faster—most of them can—but a good programmer knows when to leave well enough alone.

Usually we optimize for time (that is, we make the program run as quickly as possible), but sometimes a program appears to use an excessive amount of memory. There is generally a trade-off between memory and time. You can reduce memory usage by using a slower algorithm.

Guido van Rossum, Python's inventor, discussed optimizing one particular function. Take a look at that here. This algorithm shows just how many different approaches there are to solve a single problem.

Where to Optimize

Faced with an under-performing program, you first need to determine which parts of the program are causing the issues. In order to do that, you'll need to "profile" your program, that is, to find out how much time is being spent in each part of the program. This will allow you to see which pieces are taking up the most CPU time. These pieces will then be the primary targets for optimization. The Python language includes a profile module that enables you to gather detailed information about how much time is being spent in different parts of your program.

You can determine which pieces of code run faster using the facilities of the **timeit** module. For our purposes, you'll be using just a few features of the library, but I encourage you to investigate the Python library documentation outside the labs to learn more about it. Also, try out your own versions of code to learn more about different approaches to a given problem and how well they perfrom.

The Profile Module

The profile module allows you to trace your program, by keeping information about the function call and return events, as well as exceptions that are raised. It can provide detailed explanations of where your program is spending its time. The module collects and summarizes data about the various function calls in a program.

Two Different Modules

The cProfile module (written in C) functions just like the profile module, only faster. cProfile is not available in every computer's Python, though. When cProfile is unavailable, use the profile module instead. You can allow your program to make use of cProfile when it is available, and profile when it is not. A quick illustration will help you understand these tools. Here's how to import one of two modules with the same name:

OBSERVE:

```
try:
    import cProfile as profile
except ImportError:
    import profile
```

If cProfile is available, it is imported under the name profile. If it isn't available, the attempt to import it raises an ImportError exception, and the profile module is imported instead.

Using the Profile Module

Create a new Pydev project named **Python4_Lesson05**, assign it to the **Python4_Lessons** working set, and then create a new file in your **Python4_Lesson05/src** folder named **prfl.py**, as shown below:

CODE TO TYPE:

```
def f1():
    for i in range(300):
        f2()

def f2():
    for i in range(300):
        f3()

def f3():
    for i in range(300):
        pass

import cProfile as profile
profile.run("f1()")
```

The profile.run() function takes as its argument, a string containing the code to be run, and then runs it with profiling active. If only one argument is given, the function produces output at the end of the run that summarizes the operation of the code.

- Save and run it; you see something like this:

OBSERVE:

```
         90304 function calls in 1.110 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.110    1.110 <string>:1(<module>)
        1    0.000    0.000    1.110    1.110 prfl2.py:1(f1)
      300    0.030    0.000    1.110    0.004 prfl2.py:5(f2)
    90000    1.080    0.000    1.080    0.000 prfl2.py:9(f3)
        1    0.000    0.000    1.110    1.110 {built-in method exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof
iler' objects}
```

A total of 90304 function calls are recorded during the execution of the code in a total of 1.110 seconds. The rest of the output is sorted by function name by default. The columns are:

Column Name	Meaning
ncalls	The total number of calls made to the listed function.
tottime	The total time spent in executing the listed function.
percall (1)	The average execution time for a single call of the function.
cumtime	The cumulative execution time of all calls of this function, including the time taken to execute all functions called from this one.
percall (2)	The average cumulative execution time for a single call of the function.

filename:lineno(function)	The details of the source code defining the function..

By looking at the "tottime" column, we can see that the majority of the program's time is spent in the f3() function. In fact, if you could eliminate the time taken by the rest of the program altogether, the impact to the program's total execution time would be less than 5%. In other words, the f3() function is taking up 95% of the program's execution time. As Guido van Rossum says:

Rule number one: only optimize when there is a proven speed bottleneck. Only optimize the innermost loop. (This rule is independent of Python, but it doesn't hurt repeating it, since it can save a lot of work.) :-)

More Complex Reporting

Sometimes you'll want more specific information from a profiling run. When that's the case, you'll use the second argument to profile.run—the name of a file to which your program will send the raw profiling data. Then you can process this data separately using the pstats module. In order to give the module enough data to work with, we'll use another artificially constructed program (there is no real computation taking place, but many function calls). Modify **prfl.py** to add more function calls:

CODE TO TYPE:

```
def f1():
    for i in range(300):
        f2(); f3(); f5()

def f2():
    for i in range(300):
        f3()

def f3():
    for i in range(300):
        pass

def f4():
    for i in range(100):
        f5()

def f5():
    i = 0
    for j in range(100):
        i += j
    f6()

def f6():
    for i in range(100):
        f3()

import cProfile as profile
profile.run("f1()", "profiledata")
```

When you run this program, you won't see any output in the console window. The program creates a file named **profiledata** in the folder where **prfl.py** is located (refresh the Package Explorer window [press **F5**] to see it). Now if you start up a console window in the same directory (make sure the program is in the active editor window, select **PyDev Console** from the **Open Console** pull-down menu, select **Console for currently active editor**, then click **OK**), you can work with that file using the pstats module, written precisely to allow analysis of the profile data.

The primary element in the pstats module is the Stats class. When you create an instance, you can give it the name(s) of one or more files as positional arguments. These files will have been created by profiling. You can also provide a stream keyword argument, which is an open file to which output will be sent (this defaults to standard output, meaning you see the output straight away).

Note The next series of operations should all be performed in the same console window, so do not close it down between operations.

Make sure to keep this window open after this interactive session:

INTERACTIVE CONSOLE SESSION

```
>>> import pstats
>>> s = pstats.Stats("V:\\workspace\\Python4_Lesson05\\src\\profiledata")

>>> s.print_stats()
Mon Jun 25 17:55:43 2012    V:\workspace\Python4_Lesson05\src\profiledata

         121204 function calls in 3.275 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.275    3.275 {built-in method exec}
      300    0.770    0.003    2.458    0.008 V:\workspace\Python4_Lesson05\src\
prfl.py:5(f2)
      300    0.259    0.001    0.795    0.003 V:\workspace\Python4_Lesson05\src\
prfl.py:23(f6)
        1    0.007    0.007    3.275    3.275 V:\workspace\Python4_Lesson05\src\
prfl.py:1(f1)
        1    0.000    0.000    3.275    3.275 <string>:1(<module>)
   120300    2.229    0.000    2.229    0.000 V:\workspace\Python4_Lesson05\src\
prfl.py:9(f3)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof
iler' objects}
      300    0.010    0.000    0.804    0.003 V:\workspace\Python4_Lesson05\src\
prfl.py:17(f5)


<pstats.Stats object at 0x0000000002955198>
>>>
```

Note The times and paths in your output will vary from the values in the above console session.

When you create a pstats.Stats instance, it loads the data, and you can manipulate it before producing output (you'll see how shortly). There are several refinements you can make to the output, by calling methods of your Stats instance.

INTERACTIVE CONSOLE SESSION

```
>>> s.strip_dirs()   # shorten function references
<pstats.Stats object at 0x0000000002955198>
>>> s.print_stats()
Mon Jun 25 17:55:43 2012    V:\workspace\Python4_Lesson05\src\profiledata

         121204 function calls in 3.275 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.275    3.275 {built-in method exec}
        1    0.007    0.007    3.275    3.275 prfl.py:1(f1)
   120300    2.229    0.000    2.229    0.000 prfl.py:9(f3)
      300    0.259    0.001    0.795    0.003 prfl.py:23(f6)
      300    0.770    0.003    2.458    0.008 prfl.py:5(f2)
        1    0.000    0.000    3.275    3.275 <string>:1(<module>)
      300    0.010    0.000    0.804    0.003 prfl.py:17(f5)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof
iler' objects}


<pstats.Stats object at 0x0000000002955198>
>>>
```

The **strip_dirs()** method has removed all of the directory information from the last column. **strip_dirs()** is applied to the default output; the path information isn't generally required. Next, you can sort the output to give you the most significant items first by providing one or more keys to the **Stats.sort_stats()** method. The keys that are acceptable currently are:

Key	Sort by ...
'calls'	The total count of calls of the function (including "recursive calls" where a function calls itself, or calls other functions which in turn call it).
'cumulative'	Cumulative execution time
'file'	File name from which the function was loaded
'module'	Same as **'file'**
'pcalls'	Count of *primitive* calls (i.e. calls made to the function while it is not actually executing)
'line'	Line number
'name'	Function name
'nfl'	Name/file/line
'stdname'	Sorts by the function name as printed
'time'	Internal time

You may have noticed that 3 of the 8 lines of the output aren't particularly useful for our requirements. Fortunately, you can filter out the results you don't want by placing one or more restrictions on the output. Those restrictions can take one of three forms as additional arguments to **print_stats()**:

- An integer will limit the output to the given number of lines.
- A floating-point number between 0 and 1 will restrict the output to the given proportion of entries.
- A regular expression will limit the output to those entries whose filename:lineno(function) fields contain the given regular expression.

You can limit the output to omit the details of the "structural" entries (those that relate strictly to the profiling framework) using the simple expression **r"\.py"**, or, once the entries are sorted in the right order, by using the integer 5 in this case.

The restrictions are applied in order, so **print_stats(0.1, "test")** reports those lines out of the top tenth that match "test", whereas **print_stats("test", 0.1)** reports a tenth of all those lines matching "test." So, if there

were a hundred lines in the source data, **print_stats(0.1, "test")** would print any lines that contain "test" from the first ten. **print_stats("test", 0.1)** would print one tenth of ALL the lines that contain "test." If every third line contained "test", **print_stats(0.1, "test")** would retrieve lines 3,6, and 9. **print_stats("test", 0.1)** would retrieve lines 3,6,9, and 11 -- four lines (assuming there were about 40 containing "test").

INTERACTIVE CONSOLE SESSION

```
>>> s.sort_stats('calls', 'time')
<pstats.Stats object at 0x10057c510>
>>> s.print_stats(r"\.py")
Mon Jun 25 17:55:43 2012    V:\workspace\Python4_Lesson05\src\profiledata

         121204 function calls in 3.275 seconds

   Ordered by: call count, internal time
   List reduced from 8 to 5 due to restriction <'\\.py'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   120300    2.229    0.000    2.229    0.000 prfl.py:9(f3)
      300    0.770    0.003    2.458    0.008 prfl.py:5(f2)
      300    0.259    0.001    0.795    0.003 prfl.py:23(f6)
      300    0.010    0.000    0.804    0.003 prfl.py:17(f5)
        1    0.007    0.007    3.275    3.275 prfl.py:1(f1)


<pstats.Stats object at 0x0000000002955198>
>>> s.print_stats(5)
Mon Jun 25 17:55:43 2012    V:\workspace\Python4_Lesson05\src\profiledata

         121204 function calls in 3.275 seconds

   Ordered by: call count, internal time
   List reduced from 8 to 5 due to restriction <5>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   120300    2.229    0.000    2.229    0.000 prfl.py:9(f3)
      300    0.770    0.003    2.458    0.008 prfl.py:5(f2)
      300    0.259    0.001    0.795    0.003 prfl.py:23(f6)
      300    0.010    0.000    0.804    0.003 prfl.py:17(f5)
        1    0.007    0.007    3.275    3.275 prfl.py:1(f1)


<pstats.Stats object at 0x0000000002955198>
>>>
```

Note

You may have wondered why all of the methods of the **pstats.Stats** object seem to return the same **pstats.Stats** instance. It's to allow users to utilize a technique called *method chaining*. Since each method call returns the instance, you can apply a method call directly to the result of a previous method call, as in

s.strip_dirs().sort_stats('calls','time').print_stats()

You'll also want to know which functions call which other functions. The pstats.Stats object has the print_callers() and print_callees() methods that show you the calling relationships between various functions:

INTERACTIVE CONSOLE SESSION

```
>>> s.sort_stats('calls', 'time')
<pstats.Stats object at 0x0000000002955198>

>>> s.print_callers(r"\.py")
   Ordered by: call count, internal time
   List reduced from 8 to 5 due to restriction <'\\.py'>

Function        was called by...
                    ncalls  tottime  cumtime
prfl.py:9(f3)   <-     300    0.005    0.005  prfl.py:1(f1)
                     90000    1.688    1.688  prfl.py:5(f2)
                     30000    0.536    0.536  prfl.py:23(f6)
prfl.py:5(f2)   <-     300    0.770    2.458  prfl.py:1(f1)
prfl.py:23(f6)  <-     300    0.259    0.795  prfl.py:17(f5)
prfl.py:17(f5)  <-     300    0.010    0.804  prfl.py:1(f1)
prfl.py:1(f1)   <-       1    0.007    3.275  <string>:1(<module>)


<pstats.Stats object at 0x0000000002955198>
>>> s.print_callees(r"\.py")
   Ordered by: call count, internal time
   List reduced from 8 to 5 due to restriction <'\\.py'>

Function         called...
                     ncalls  tottime  cumtime
prfl2.py:9(f3)   ->
prfl2.py:5(f2)   ->   90000    1.080    1.080  prfl2.py:9(f3)
prfl2.py:23(f6)  ->   30000    0.355    0.355  prfl2.py:9(f3)
prfl2.py:17(f5)  ->     300    0.010    0.365  prfl2.py:23(f6)
prfl2.py:1(f1)   ->     300    0.027    1.107  prfl2.py:5(f2)
                        300    0.004    0.004  prfl2.py:9(f3)
                        300    0.004    0.369  prfl2.py:17(f5)


<pstats.Stats object at 0x0000000002955198>
>>>
```

Being aware of which function calls which other functions can be useful when you are trying to locate specific calls that take more time than others.

What to Optimize

You can use the profile module to hone in on the parts of your program that are using the most CPU time. Your next consideration will be figuring out how to speed up the code in your "hot spots." To do this, we'll use the timeit module, which allows you to measure the relative speeds of different Python snippets. The timeit module contains more features than we need for our task, but it's a good idea to familiarize yourself with its documentation for future tasks.

The timeit module defines a Timer class which allows you full control over the creation and execution of timed code, but we'll just use the module's timeit() function; it allows you to specify a statement to be timed and some initialization code to execute before timing starts. The function runs the initialization code and then executes the code to be timed repeatedly, printing out the total execution time in seconds. Take a look:

INTERACTIVE CONSOLE SESSION

```
>>> from timeit import timeit
>>> timeit("i = i + 1", "i=0")
0.11318016052246094
>>> timeit("i = i + 1", "i=0")
0.11426806449890137
>>> timeit("i = i + 1", "i=0")
0.1136329174041748
>>> timeit("i += 1", "i=0")
0.11641097068786621
>>> timeit("i += 1", "i=0")
0.11541509628295898
>>> timeit("i += 1", "i=0")
0.11439919471740723
>>>
```

The example demonstrates that timings are not completely repeatable (and therefore shouldn't be relied upon for absolute information). Secondly, it demonstrates that there isn't a big difference between the time it takes to execute regular addition and the time required to execute the augmented addition operator.

Note The timeit() function creates an entirely new namespace in which to run the code being timed, so the examples use an initialization statement to set **i** to zero before the timed code is run; without that, you'd see an exception indicating that the **i** had not been defined.

Now that you know how modules work, we can concentrate on getting your code to run faster. To help facilitate writing your timing tests, you'll usually define functions containing the code that are called by the timing routine.

Loop Optimizations

Sometimes you write code and put a computation inside of the loop when it doesn't need to be. Under those circumstances there are gains to be made by moving the computation out of the loop, a technique usually referred to as "loop hoisting." Here is an example of loop hoisting:

INTERACTIVE CONSOLE SESSION

```
>>> def loop1():
...     lst = range(10)
...     for i in lst:
...         x = float(i)/len(lst)
...
>>> def loop2():
...     lst = range(10)
...     ln = len(lst)
...     for i in lst:
...         x = float(i)/ln
...
>>> timeit("loop1()", "from __main__ import loop1")
7.349833011627197
>>> timeit("loop2()", "from __main__ import loop2")
4.197483062744141
>>>
```

What seems like a small change to the code makes a substantial difference!

Actually, the best way to optimize a loop is to remove it altogether. Sometimes you can do that using Python's built-in functions. Let's time four different ways to build the upper-case version of a list:

INTERACTIVE CONSOLE SESSION

```
>>> oldlist = "the quick brown fox jumps over the lazy dog".split()
>>> def lf1(lst):
...     newlist = []
...     for w in lst:
...         newlist.append(w.upper())
...     return newlist
...
>>> def lf2(lst):
...     return [w.upper() for w in lst]
...
>>> def lf3(lst):
...     return list(w.upper() for w in lst)
...
>>> def lf4(lst):
...     return map(str.upper, lst)
...
>>>
>>> timeit("lf1(oldlist)", "from __main__ import lf1, oldlist")
4.409790992736816
>>> timeit("lf2(oldlist)", "from __main__ import lf2, oldlist")
3.492004156112671
>>> timeit("lf3(oldlist)", "from __main__ import lf3, oldlist")
4.758850812911987
>>> timeit("lf4(oldlist)", "from __main__ import lf4, oldlist")
0.5220911502838135
>>>
```

You haven't run into the map() buit-in before, but it has some good things going for it. Its first argument is a function (in this case, the unbound upper() method of the built-in str type), and any remaining arguments are iterables. There are as many iterables as the function takes arguments, and the result is a list containing the return values of the function when called with corresponding elements of each iterable (if the iterables are not all the same length, map stops as soon as the first one is exhausted).

So, why is the map()-based solution so much faster? There are two reasons. First, it is the only solution that does not need to look up the upper() method in the str type each time around the loop. Second, the looping is done inside map(), which is written in the C language, which saves a lot of time.

Another way to remove a loop is to write the loop contents out as literal code. This is really only practical for short loops with a known number of iterations, but it can be a very effective technique, as the next example of "inlining loop code" shows:

INTERACTIVE CONSOLE SESSION

```
>>> def f1():
...     pass
...
>>> def loopfunc():
...     for i in range(8):
...         f1()
...
>>> def inline():
...     f1(); f1(); f1(); f1(); f1(); f1(); f1(); f1()
...
>>> timeit("loopfunc()", "from __main__ import loopfunc")
1.9027259349822998
>>> timeit("inline()", "from __main__ import inline")
1.2639250755310059
>>>
```

There can be a substantial amount of overhead in looping. When function calls are written out explicitly, the execution time is 30% faster—a worthwhile gain. Of course, in this example the loop overhead does tend to

dominate because there is so little actual computation happening.

Pre-computing Attribute References

Due to Python's dynamic nature, when the interpreter comes across an expression like **a.b.c**, it looks up *a* (trying first the local namespace, then the global namespace, and finally the built-in namespace), then it looks in *that* object's namespace to resolve the name *b*, and finally it looks in *that* object's namespace to resolve the name *c*. These lookups are reasonably fast; for local variables, lookups are extremely fast, since the interpreter knows which variables are local and can assign them a known position in memory. There are definitely gains to be had by storing references in local variables. Let's try removing Attribute Resolution from loops:

INTERACTIVE CONSOLE SESSION

```
>>> class Small:
...     class Smaller:
...         x = 20
...     smaller = Smaller
...
>>> small = Small()
>>>
>>> def attr1():
...     ttl = 0
...     for i in range(50):
...         ttl += small.smaller.x
...     return ttl
...
>>> def attr2():
...     ttl = 0
...     x = small.smaller.x
...     for i in range(50):
...         ttl += x
...     return ttl
...
>>> timeit("attr1()", "from __main__ import small, attr1")
11.901235103607178
>>> timeit("attr2()", "from __main__ import small, attr2")
6.448068141937256
>>>
```

Here, the function doesn't actually execute a huge amount of computation, but we gain a lot in speed.

Local Variables are Faster than Global Variables

As we mentioned before, the interpreter knows which names inside your functions are local and it assigns them specific (known) locations inside the function call's memory. This makes references to locals much faster than to globals and (most especially) to built-ins. Let's test name reference speed from various spaces:

INTERACTIVE CONSOLE SESSION

```
>>> glen = len # provides a global reference to a built-in
>>>
>>> def flocal():
...     name = len
...     for i in range(25):
...         x = name
...
>>> def fglobal():
...     for i in range(25):
...         x = glen
...
>>> def fbuiltin():
...     for i in range(25):
...         x = len
...
>>> timeit("flocal()", "from __main__ import flocal")
1.743438959121704
>>> timeit("fglobal()", "from __main__ import fglobal")
2.192162036895752
>>> timeit("fbuiltin()", "from __main__ import fbuiltin")
2.259413003921509
>>>
```

This difference in speed isn't huge here, but it definitely shows that accessing a local variable is faster than accessing a global or a built-in. If many globals or built-ins are used inside a function, it makes sense to store a local reference to them. By contrast, if they are used only once, then you'd only be adding overhead to your function!

How to Optimize

Optimizing code isn't easy, and it would be impossible to show you all the gotchas you can introduce into your code here. For now, here are a few guidelines that can help you avoid common pitfalls.

Don't Optimize Prematurely

Don't consider performance while you're writing the code (although it's difficult for even experienced programmers to ignore). The primary goal of the initial programming process is a correct, functioning algorithm that is relatively easy to understand. Only after your tests demonstrate correct operation should you address performance.

Use Timings, Not Intuition

Our intuition is not always the best gauge of what will run fast. You're much better off using timings to determine how well your program is running.

Make One Change at a Time

If you make two changes to a program, and the first makes a 10% improvement, that's great, right? But if the second takes performance *down* by 25%, the overall result will be worse than those of the unchanged program. Make your changes individually and methodically.

The Best Way is Not Always Obvious

Guido van Rossum has yet more wisdom to share with us (I am a fan). In the article we mentioned above he presents us with a problem: given a list of integers in the range 0-127 (these are ASCII values; Python 2 was current when Guido wrote this), how does one create a string in which the characters have the ordinal values held in the corresponding positions in the list of integers? Guido (I think we have spent enough quality time with Guido to be on a first name basis now) realized that the fastest way to create such a string was to take advantage of the array module's ability to create one-byte integers; he came up with this code:

OBSERVE:

```
import array
def f7(list):
    return array.array('B', list).tostring()
```

When you are writing code, the obvious way is the best. To extract maximum performance the best way is not always obvious! Did I really say this was a short lesson? Time flies when we're deep into the Python! You're doing really well so far.

When you finish the lesson, don't forget to return to the syllabus and complete the homework.

Using Exceptions Wisely

Lesson Objectives

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

- identify which exceptions are errors.
- create exceptions and raise instances.
- use exceptions wisely.

Exceptions Are Not (Necessarily) Errors

Raising an exception alters the flow of control in a program. The interpreter normally executes statements one after the other (with looping to provide repetition, and conditionals to allow decision-making). When an exception is raised, however, an entirely different mechanism takes over. Precisely because it *is* exceptional, we tend to be less familiar with it, but knowing how exceptions are raised and handled can help you to program to focus on the main task, in confidence that when exceptional conditions do occur, they will be handled appropriately. Knowing how, and when, to use exceptions is a part of your development as a Python programmer.

Exceptions offer such programming convenience that we would likely be quite happy to pay a modest penalty in performance. The happy fact is, though, that when used judiciously exceptions can actually enhance your programs' performance as well as making them easier to read.

Specifying Exceptions

Python's built-in exceptions are all available (in the built-in namespace, naturally) without any import. There is an inheritance hierarchy among them. From the Python documentation:

Python's Built-In Exception Hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Although everything inherits from the **BaseException** class, its first three subclasses (**SystemExit**, **KeyboardInterrupt** and **GeneratorExit**) should not be caught and handled by regular programs under normal circumstances. About the most general specification to catch would normally be **except Exception**,

and that would be reserved for programs such as long-running network servers or equipment control and monitoring applications.

The full syntax of the **except** clause allows you to specify not just a single exception but a whole class or set of them, all to be handled in the same way by the same except clause. When you specify an exception class then, any of its subclasses will also be caught (unless, that is, the subclass is in an earlier except clause for the same try and therefore caught already). In other words, if your program catches **ArithmeticError**, it also catches **FloatingPointError**, **OverflowError** and **ZeroDivisionError**. As the next interactive session should make plain, under some circumstances the ordering of the except clauses will make a difference in which handler handles the exception.

Where subclasses are concerned, except clause ordering is significant

```
>>> try:
...     raise ZeroDivisionError
... except ArithmeticError:
...     print("ArithmeticError")
... except ZeroDivisionError:
...     print("ZeroDivisionError")
...
ArithmeticError
>>> try:
...     raise ZeroDivisionError
... except ZeroDivisionError:
...     print("ZeroDivisionError")
... except ArithmeticError:
...     print("ArithmeticError")
...
ZeroDivisionError
>>>
```

OBSERVE:

```
try:
    raise ZeroDivisionError
except ArithmeticError:
    print("ArithmeticError")
except ZeroDivisionError:
    print("ZeroDivisionError")

ArithmeticError
try:
    raise ZeroDivisionError
except ZeroDivisionError:
    print("ZeroDivisionError")
except ArithmeticError:
    print("ArithmeticError")

ZeroDivisionError
```

In the first example, since ZeroDivisionError is a subclass of ArithmeticError, the first **except** clause is triggered, and the **ZeroDivisionError** is never tested for (since the second **except** clause was never evaluated). In the second example, the **ZeroDivisionError** is specifically recognized because it is tested for before the **ArithmeticError**.

Creating Exceptions and Raising Instances

If you want to create your own exceptions, simply subclass the built-in Exception class or one of its already existing subclasses. Then create instances as required to raise exceptions. You may want to include an __init__() method on your subclass. The standard Exception.__init__() saves the tuple of positional arguments to the args attribute, so you can either do the same yourself or call Exception.__init__() to do it on your behalf. Your exceptions may at some stage be passed to a piece of code that expects to find an args instance variable.

Here's an example of a user-defined exception.

How to Define an Exception [keep this session open and re-use it]

```
>>> class LocalError(Exception):
...     def __init__(self, msg):
...         self.args = (msg, )
...         self.msg = msg
...     def __str__(self):
...         return self.msg
...
>>> try:
...     raise LocalError("Appropriate message")
... except LocalError as e:
...     print("Trapped", e)
...
Trapped Appropriate message
>>> raise LocalError
Traceback (most recent call last):
  File "<sonsole>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument: 'msg'>>>
```

This exception class requires an argument when an instantiation call is made to create a new instance—without one, the __init__() method does not receive enough arguments. You can see this happening when the **raise LocalError** statement is executed at the end of the session: when you use a class to raise an exception, the interpreter attempts to create an instance of that exception by calling the class with no arguments. So the message you see has nothing to do with the exception you have tried to raise; it's reporting the interpreter's inability to create an exception instance because of an argument mismatch in the __init__() method.

Exception objects are generally simple—the most they normally do is establish attribute values that can be used by the handler to extract information about the exception. Since they are classes, it is possible to add complex logic in multiple methods, but this is normally not done. As usual in Python, simplicity is the order of the day.

Understanding the straightforward flow of control when an exception is raised in the try suite is relatively easy. It is less easy to appreciate what happens when exceptions occur in the except or finally suites. To look at that, define a function that raises exceptions in one of those three places, then see what it does under those circumstances.

Create a new PyDev project named **Python4_Lesson06** and assign it to the **Python4_Lessons** working set. Then, in your **Python4_Lesson06/src** folder, create **fxfin.py** as shown:

CODE TO TYPE: Create the following file as fxfin.py

```
class LocalError(Exception):
    def __init__(self, msg):
        self.args = (msg, )
        self.msg = msg
    def __str__(self):
        return self.msg

def fxfin(where):
    "Demonstrate exceptions in various places."
    try:
        if where == "try":
            raise LocalError("LocalError in try")
        raise ValueError("ValueError in try")
    except (ValueError, LocalError) as e:
        print("Caught", e)
        if where == "except":
            raise LocalError("LocalError in except")
        print("Exception not raised in except")
    finally:
        print("Running finalization")
        if where == "finally":
            raise LocalError("LocalError in finally")
        print("Exception not raised in finally")

for where in "try", "except", "finally":
    print("---- Exception in %s ----" % where)
    try:
        fxfin(where)
    except Exception as e:
        print("!!!", e, "raised")
    else:
        print("+++ No exception raised +++")
```

- Run the program; you see the following output:

Results of running fxfin.py

```
---- Exception in try ----
Caught LocalError in try
Exception not raised in except
Running finalization
Exception not raised in finally
+++ No exception raised +++
---- Exception in except ----
Caught ValueError in try
Running finalization
Exception not raised in finally
!!! LocalError in except raised
---- Exception in finally ----
Caught ValueError in try
Exception not raised in except
Running finalization
!!! LocalError in finally raised
```

When the exception is raised in the try suite, everything is perfectly normal and comprehensible, and both the except and finally handlers run without interruption. By the time the finally suite runs the exception has already been fully handled. The except suite is always activated, but it can be so either by virture of the parameter value or because of the final explicit exception. This means the except clause is more readable. With the "except" argument the handler raises a second exception. This terminates the except handler, but the finally handler still runs; once it is complete, the second exception is still raised from the function. When the exception is raised in the finally suite, the finally handler does not run to completion, and the exception is passed up to the surrounding code (so the traceback is produced because of an uncaught exception).

Note that when you see a traceback for the case where an exception is raised during the handling of an exception that a

second exception occurred during the processing of the first. This information may be confusing to end users, but can be invaluable to a programmer.

Using Exceptions Wisely

Let's take a look at the bytecodes that the CPython 3.1 interpreter produces for a simple function with exception handling.

Note Different Python interpreters may use entirely different techniques to handle exceptions, but the effect should always be the same as in these descriptions.

Examine the CPython byte code for try/except

```
>>> import dis
>>> def fex1():
...     try:
...         a = 1
...     except KeyError:
...         b = 2
...
>>> dis.dis(fex1)
  2           0 SETUP_EXCEPT             10 (to 13)

  3           3 LOAD_CONST               1 (1)
              6 STORE_FAST               0 (a)
              9 POP_BLOCK
             10 JUMP_FORWARD             24 (to 37)

  4     >>   13 DUP_TOP
             14 LOAD_GLOBAL              0 (KeyError)
             17 COMPARE_OP              10 (exception match)
             20 POP_JUMP_IF_FALSE       36
             23 POP_TOP
             24 POP_TOP
             25 POP_TOP

  5          26 LOAD_CONST               2 (2)
             29 STORE_FAST               1 (b)
             32 POP_EXCEPT
             33 JUMP_FORWARD             1 (to 37)
        >>   36 END_FINALLY
        >>   37 LOAD_CONST               0 (None)
             40 RETURN_VALUE
>>>
```

The interpreter establishes an exception-handling context by pointing at location 13 as the place to go if an exception occurs (this is what the SETUP EXCEPT op code does). This is followed by the body of the try clause. If the try clause reaches the end, the POP_BLOCK opcode throws away the exception-handling context and the JUMP_FORWARD sends the interpreter off to perform the implicit **return None** that terminates every function.

If an exception is raised, however, control is transferred to location 13, where the interpreter attempts to match the exception to the except specifications. If a match is found (and after various housekeeping operations we will ignore), line 26 is where the except suite is performed, after which another JUMP_FORWARD again selects the implicit **return None**. If no match is found for the exception, the END_FINALLY ensures that the exception is re-raised to activate any surrounding exception-handling contexts.

The try/except blocks in your program can be nested *lexically* (that is, a try/except can be a part of the try suite of another try suite) or *dynamically* (that is, a try suite can call a function that activates one or more try/excepts). When a try block is nested dynamically, it will be deactivated by termination of the function *even if the return statement is in the try suite or an except suite*. The finally suite is *always* executed, even when the function returns from an unexpected place. An explicit return in the finally suite does not allow that suite to run to completion—instead the return is executed (overriding any return value that might have triggered the execution of the finally clause).

Exception Timings

Sometimes in optimization, it's useful to be able to know how "expensive" it is in time to handle an exception. With judicious coding, you can actually save time using exceptions, but you (as always) need to think about what you are doing rather than just applying rules blindly. The next interactive session shows that it can be good or bad to rely on exceptions, depending on the surrounding circumstances.

Exception timings depend on how frequently the exception is raised

```
>>> def fdct1():
...     wdict = {}
...     for word in words:
...         if word not in wdict:
...             wdict[word] = 0
...         wdict[word] += 1
...
>>> def fdct2():
...     wdict = {}
...     for word in words:
...         try:
...             wdict[word] += 1
...         except KeyError:
...             wdict[word] = 1
...
>>> from timeit import timeit
>>> words = "the quick brown fox jumps over the lazy dog".split()
>>> timeit("fdct1()", "from __main__ import fdct1")
4.041514158248901
>>> timeit("fdct2()", "from __main__ import fdct2")
6.705680847167969
>>> words = ["same"] * 9
>>> timeit("fdct1()", "from __main__ import fdct1")
2.6857001781463623
>>> timeit("fdct2()", "from __main__ import fdct2")
2.948345899581909
>>>
```

Here you did two sets of timings, the first with a word list in which there was only one duplicate, the second with one where every word was the same. Under the former conditions the specific test for **word not in wdict** won out against raising an exception. In the second case, however, when the exception was rarely raised, the exception-based solution was at least competitive although still not actually faster. Thus, the optimal code can depend to some extent on the data. If you have advance information about the make-up of your data, that's all very well, but if not, it would be more difficult to try and choose between approaches.

The important thing is not to run away with the idea that exceptions are somehow intended to be used in exceptional circumstances. If your logic is easier to express with exceptions, use them. If for some reason your program, once working, does not work fast enough, you can refactor it (making sure you do not break any tests) for better performance.

Confidence in using exceptions to flag abnormal processing conditions is important to keep your logic simple. Without exceptions, you have to have functions return sentinel values to indicate that problems occurred during processing. With them, you can just write the logic of the main task "in a straight line" inside a try clause, and use except to catch exceptions that indicate special processing is required.

When you finish the lesson, don't forget to return to the syllabus and complete the homework.

Advanced Uses of Decorators

Lesson Objectives

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

- use decorator Syntax.
- use Classes as Decorators.
- use Class Decorators.
- employ some odd Decorator tricks.
- utilize Static and Class Method Decorators.
- parameterize Decorators.

When we discussed properties, we noted that you can use the decorator syntax to apply a function to another function. In this lesson, we'll immerse you a little more thoroughly in the uses of decoration. It can be difficult to think of small examples, however, because decorators are typically written to be applied in large systems without users having to think too deeply about it.

Decorator Syntax

Let's jump right in!

Decorator Syntax (use the same interactive session throughout this lesson)

```
>>> def trace(f):
...     "Decorate a function to print a message before and after execution."
...     def traced(*args, **kw):
...         "Print message before and after a function call."
...         print("Entering", f.__name__)
...         result = f(*args, **kw)
...         print("Leaving", f.__name__)
...         return result
...     return traced
...
>>> @trace
... def myfunc(x, a=None):
...     "Simply prints a message and arguments."
...     print("Inside myfunc")
...     print("x:", x, "a:", a)
...
>>> myfunc("ONE", "TWO")
Entering myfunc
Inside myfunc
x: ONE a: TWO
Leaving myfunc
>>>
```

In the example above, the trace function is a decorator. That means that it takes a single argument (which is normally the function being decorated). Internally, it defines a function traced() that prints out a line of text, calls the decorated function with whatever arguments it was called with itself, prints out another line of text and then returns the result obtained from the decorated function. Then, *trace returns the function it has just defined*.

This means that you can apply trace() to any function, and the result will do just what the original function did as well as printing out a line before and after the call to the decorated function. This is how most decorators work (although as always there are some smart people who have found non-standard ways to use decorators that were not originally intended by the specification). That's why you often see the internal function written to accept any combination of positional and keyword arguments—it means that the decorator can be applied to any function, no matter what its signature.

Remember, the decorator syntax is really just an abbreviation; it doesn't do anything that you couldn't do without the

syntax. When you write **@trace** before the definition for myfunc(), it's exactly equivalent to writing **myfunc = trace(myfunc)** after the function definition. The syntax was added because with longer function definitions it was often difficult to notice the reassignment to the name when it followed the function definition. The feature was restricted to functions when it was originally introduced, but now you can also decorate classes. While this is a little bit more complicated than decorating functions, it does have its uses.

Because the above decorator defines a function that contains a call to the decorated function as a part of its code (traced() in the example above), we say that the decorator *wraps* the decorated function. This has certain unfortunate side effects: mostly, the name of the function appears to change to the name of the wrapper function from inside the decorator, and the docstring is that of the wrapper.

The decorated function name differs from the undecorated one

```
>>> trace.__name__          # undecorated
'trace'
>>> myfunc.__name__         # decorated
'traced'
>>> myfunc.__doc__
'Print message before and after a function call.'
>>>
```

Fortunately, this issue can be handled using the **wraps** decorator from the **functools** library. This is provided precisely to ensure that decorated functions continue to "look like themselves." Until you get the hang of using it, however, it seems a little weird because it means you end up *using a decorator on the wrapper function inside your decorator*! But honestly, it isn't difficult.

Use functools.wraps to avoid loss of name and docstring

```
>>> from functools import wraps
>>> def simpledec(f):
...     "A really simple decorator to demonstrate functools.wraps."
...     @wraps(f)
...     def wrapper(arg):
...         print("Calling f with arg", arg)
...         return f(arg)
...     return wrapper
...
>>> @simpledec
... def f(x):
...     "Simply prints its argument."
...     print("Inside f, arg is", x)
...
>>> f("Hello")
Calling f with arg Hello
Inside f, arg is Hello
>>> f.__name__
'f'
>>> f.__doc__
'Simply prints its argument.'
>>>
```

Classes as Decorators

While decorators are usually functions, they don't need to be—any callable can be used as a decorator. This means that you could use a class as a decorator, and when the decoration takes place the class's __init__() method is called with the object to be decorated (whether it's a function or a class: note that a decorator is typically designed to decorate either functions or classes but not both because they are fairly different in nature).

If you want to decorate a function with a class, remember that calling a class calls its __init__() method, and returns an instance of the class. As always, the first argument to __init__() is *self*, the newly created instance, so in this case the function that the interpreter passes to the decorator will end up as the second argument to __init__(). Since calling the class creates an instance, and since normally you want to be able to call the decorated function, the classes you use as decorators should define a __call__() method, which will then be called when the decorated function is called.

Classes can be decorators too!

```
>>> class ctrace:
...     def __init__(self, f):
...         "__init__ records the passed function for later use in __call__()."
...         self.__doc__ = f.__doc__
...         self.__name__ = f.__name__
...         self.f = f
...     def __call__(self, *args, **kw):
...         "Prints a trace line before calling the wrapped function."
...         print("Called", self.f.__name__)
...         return self.f(*args, **kw)
...
>>> @ctrace
... def simple(x):
...     "Just prints arg and returns it."
...     print("simple called with", x)
...     return x
...
>>> simple("walking")
Called simple
simple called with walking
'walking'
>>> simple.__name__
'simple'
>>> simple.__doc__
'Just prints arg and returns it.'
>>>
```

By the time the decorator is called, the simple() function has already been compiled, and it is passed to the decorator's __init__() method, where it is stored as an instance variable. To make sure the decorated function retains its name and docstring, those attributes of the function are also copied into instance variables with the same names.

Class Decorators

Up until now, we have decorated functions, but once the feature was introduced into Python, it was only a matter of time before it was extended to classes. So now you can decorate classes in just the same way as functions. The principle is exactly the same: the decorator receives a class as an argument, and (usually) returns a class. Because classes are more complicated than functions you will find it most convenient to modify the class in place and return the modified class as the result of the decorator.

Note Decorators can be applied individually to the methods of a class. Essentially they are the same as functions, and so exactly the same techniques can be used with methods as with regular functions.

To demonstrate this, suppose that you want to be able to have each of the methods of a class print out a trace call during debugging. You could simply apply the trace decorator above to each method, but that would mean extensive editing for a large class when you wanted to switch the debugging off. It is simpler for programmers to use a class decorator, so we might well accept a slightly higher level of complexity in the decorator to avoid the editing. Once the interpreter has processed the class definition, it calls the decorator with the class as its argument, and the decorator can then either create a new class (which is fairly difficult) or modify the class and return it.

Since the interactive session has already defined a simple tracing function, we'll use that to wrap each of the methods in our decorated class. Finding the methods is not as easy as you might imagine. It involves looking through the class's __dict__ and finding callable items whose names do not begin and end with "__" (it's best not to mess with the "magic" methods). Once such an item is found, it is wrapped with the trace() function and replaced in the class __dict__.

Using a class decorator to wrap each method

```
>>> def callable(o):
...     return hasattr(o, "__call__")
...
>>> def mtrace(cls):
...     for key, val in cls.__dict__.items():
...         if key.startswith("__") and key.endswith("__") \
...                     or not callable(val):
...             continue
...         setattr(cls, key, trace(val))
...         print("Wrapped", key)
...     return cls
...
>>> @mtrace
... class dull:
...     def method1(self, arg):
...         print("Method 1 called with arg", arg)
...     def method2(self, arg):
...         print("Method 2 called with arg", arg)
...
Wrapped method2
Wrapped method1
>>> d = dull()
>>> d.method1("Hello")
Entering method1
Method 1 called with arg Hello
Leaving method1
>>> d.method2("Goodbye")
Entering method2
Method 2 called with arg Goodbye
Leaving method2
>>>
```

Note The __dict__ of a *class* (as opposed to that of an instance) isn't a plain dict like the ones you know. It is actually an object called a dict_proxy. To keep them as lightweight as possible, they do not directly support item assignment like a standard dict does. This is why, in the mtrace() function, the wrapped method replaces the original version by using the setattr() built-in function.

Note The callable() function was present by accident in 3.0. The developers had intended to remove it, thinking that it could easily be replaced by hasattr(obj "__call__"). Consequently it was removed from Python 3.1. It was then reinstated in Python 3.2 when some developers pointed out that a more specific version could be written in C with full access to the object structures.

As you can see, when you call method1() and method2(), they print out the standard "before and after" trace lines, because they are now wrapped by the trace() function.

Odd Decorator Tricks

Sometimes you don't want to wrap the function: instead you want to alter it in some other way, such as adding attributes (yes, you can add attributes to functions the same way as you can to most of the other objects in Python). In that case, the decorator simply returns the function that is passed in as an argument, having modified the function in whatever ways it needs to. So next we'll write a decorator that flags a function as part of a framework by adding a "framework" attribute.

Using a decorator to add attributes rather than wrapping a function

```
>>> def framework(f):
...     f.framework = True
...     f.author = "Myself"
...     return f
...
>>> @framework
... def somefunc(x):
...     pass
...
>>> somefunc.framework
True
>>> somefunc.author
'Myself'
>>>
```

Note that the decorator does still return a function, but since there is no need to wrap the decorated function it simply returns the function that it was passed (now resplendent with new attributes). Since this avoids a second function call, it will be slightly quicker and there is no need to use functools.wraps because the function is not being wrapped.

Static and Class Method Decorators

Python includes two built-in functions that are intended for use in decorating methods. The staticmethod() function modifies a method so that the special behavior of providing the instance as an implicit first argument is no longer applied. In fact, the method can be called on either an instance or the class itself, and it will receive only the arguments explicitly provided to the call. It becomes a *static method*. You can think of static methods as being functions that don't need any information from either their class or their instance, so they do not need a reference to it. Such functions are relatively infrequently seen in the wild.

If you want to write a method that relies on data from the class (class variables are a common way to share data among the various instances of the class) but does not need any data from the specific instance, you should decorate the method with the classmethod() function to create a *class method*. Like static methods, class methods can be called on either the class or an instance of the class. The difference is that the calls to a class method *do* receive an implicit first argument. Unlike a standard method call, though, this first argument is the *class* that the method was defined on rather than the instance it was called on. The conventional name for this argument is *cls*, which makes it more obvious that you are dealing with a class method.

You may well ask what static and class methods are for—why use them when we already have standard methods that are perfectly satisfactory for most purposes? Why not just use functions instead of static methods, since no additional arguments are provided? The answer to this question lies in the fact that these functions *are* methods of a class, and so will be inherited (and can be overridden or extended) by any subclasses you may define. Further, the instances of the class can reference class variables rather than using a global—this is always safer because there is no guarantee, when your code lands in someone else's program, that their code isn't using the same global name for some other purpose. It is difficult to think of any example where the use of a classmethod would be absolutely required, but sometimes it can simplify your design a little.

A typical application for class methods has each of the instances using configuration data that is common to all, and saved in the class. If you provide methods to alter the configuration data (for example, changing the frequency a wireless transmitter works on, or changing the function that the instances call to allocate resources), they do not need to reference any of the instances, so a class method would be ideal.

Parameterizing Decorators

Sometimes you want to write a decorator that takes parameters. Remember, though, that the decorator syntax requires a callable that takes precisely one argument (the class or function to be decorated). So if you want to parameterize a decorator, you have to do so "at one remove"—the function that takes the arguments has to return a function that takes one argument and returns the decorated object. This can be a little brain-twisting, so an example may help. Or, it may just make your head explode!

Suppose that you wanted to have your program record the number of calls that are made to each of several different types of function. When you define a function, you want to give a parameter to the decorator to specify the classification of the decorated function.

Required decorator syntax to count function f as a 'special' function

```
@countable('special')
def f(...):
    ...
```

In other words, @countable('special') has to return a function that is a conventional decorator—it takes a single function as an argument and returns the decorated version of the function as its result. This means that we need to nest functions three levels deep! We will use a global variable to store a dict, and the different function-type strings will be the keys. Here we go!

Using a parameterized decorator

```
>>> counts = {}
>>> def countable(ftype):
...     "Returns a decorator that counts each call of a function against ftype."
...     def decorator(f):
...         "Decorates a function and to count each call."
...         def wrapper(*args, **kw):
...             "Counts every call as being of the given type."
...             try:
...                 counts[ftype] += 1
...             except KeyError:
...                 counts[ftype] = 1
...             return f(*args, **kw)
...         return wrapper
...     return decorator
...
>>> @countable("short")
... def f1(a, b=None):
...     print("f1 called with", a, b)
...
>>> @countable("f2")
... def f2():
...     print("f2 called")
...
>>> @countable("short")
... def f3(*args, **kw):
...     print("f3 called:", args, kw)
...
>>> for i in range(10):
...     f1(1)
...     f2()
...     f3(i, i*i, a=i)
...
f1 called with 1 None
f2 called
f3 called: (0, 0) {'a': 0}
f1 called with 1 None
f2 called
f3 called: (1, 1) {'a': 1}
f1 called with 1 None
f2 called
f3 called: (2, 4) {'a': 2}
f1 called with 1 None
f2 called
f3 called: (3, 9) {'a': 3}
f1 called with 1 None
f2 called
f3 called: (4, 16) {'a': 4}
f1 called with 1 None
f2 called
f3 called: (5, 25) {'a': 5}
f1 called with 1 None
f2 called
f3 called: (6, 36) {'a': 6}
f1 called with 1 None
f2 called
f3 called: (7, 49) {'a': 7}
f1 called with 1 None
f2 called
f3 called: (8, 64) {'a': 8}
f1 called with 1 None
f2 called
f3 called: (9, 81) {'a': 9}
>>> for k in sorted(counts.keys()):
...     print(k, ":", counts[k])
...
```

```
f2 : 10
short : 20
>>>
```

As you can see, f1 and f3 are classified as "short", while f2 is classified as "f2." Every time a @countable function is called, one is added to the count for its category. There were 30 function calls in all, 20 to category "short" (f1 and f3). Calling countable() returns a decorator whose action is to add one to the count identified by its argument. Your code defines a function (countable()) that defines a function (decorator(), which *is* a decorator, that defines a function (wrapper) that wraps the function f provided as an argument to **decorator**, which was produced by calling **countable**. This is probably about as far as anyone wants to go with decorators (and a little bit further than most).

In this survey of decorators, you can appreciate that decorators enable you to perform arbitrary manipulations of the functions and classes that you write as you write them. Decorators can, of course, also be used (though without the decorator syntax), though you should exercise extreme caution in doing so. This practice, when applied to "black box" code (code for which you have no course, and no knowledge of internal structure) is called "monkey patching", and is not generally well regarded as a production technique. But it can be valuable during experimentation.

When you finish the lesson, don't forget to return to the syllabus and complete the homework.

Advanced Generators

Lesson Objectives

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

- explain what generators represent.
- use infinite sequences.
- use the Itertools Module.
- use Generator Expressions.

What Generators Represent

Generators were added to Python to allow computation with sequences without having to actually build a data structure to hold the values of the sequence. This can yield large savings in memory. Earlier you saw that generators obey the same iteration protocol that other iterators do, and that you can write generator functions and generator expressions to avoid the creation of such sequences.

You can also use generators as "filters," to remove some of the values from an input sequence. The general pattern of such a filter is:

The General Form of a Sequence Filter

```
def filter(s):
    for v in s:
        if some_condition_on(v):
            yield v
```

This technique can easily be used to "stack" filters, by providing one filter as the argument to another. To demonstrate this technique, suppose that you wanted to examine a file, ignoring blank lines and lines beginning with a "#." While there are several ways to do this, it would be fairly simple to use generators (remembering that text files are generators too, in Python). Create a **Python4_Lesson08** project and assign it to your **Python4_Lessons** working set. Then, in the **Python4_Lesson08** project, create **filterfile.py** as follows.

filterfile.py: Using generators to filter the contents of a file

```
"""
Filter file contents using a sequence of generators.
"""
def nocomment(f):
    "Generate the non-comment lines of a file."
    for line in f:
        if not line.startswith("#"):
            yield line

def nospaces(f):
    "Generate the lines of a file without leading or trailing spaces."
    for line in f:
        yield line.strip()

def noblanks(f):
    "Generate the non-blank lines of a file."
    for line in f:
        if line:
            yield line

if __name__ == "__main__":
    for line in nocomment(noblanks(nospaces(open("py08-01.txt")))):
        print(line)
```

Now, create **py08-01.txt** as shown:

CODE TO TYPE: py08-01.txt

```
# Excluded because a comment.
# This is also a comment, and the next two lines are blank.


This line should be the first of four lines in the output.
# The next line contains spaces and tabs, and should not appear.

And this should be the second.

        # This should not appear (leading spaces but a comment).
# Neither should this (leading tabs but a comment).
        This should be the third line of output.

And this should be the last.
```

- Save the files and run filterfile.py:

Expected output from the filterfile.py

```
This line should be the first of four lines in the output.
And this should be the second.
This should be the third line of output.
And this should be the last.
```

The essence of this program is the for loop guarded by the **if __name__ == "__main__":** condition. **open("py08-01.txt")** is used to generate the raw text lines from the file, then the **nospaces()** generator strips the spaces from the lines, after which the noblanks() generator removes blank lines, and then finally the nocomment() generator yields only the lines that aren't comments.

Each individual filter performs a very simple task, but used in combination they can be much more powerful. (This is the philosophy behind the UNIX operating system, by the way: provide simple primitive commands but allow them to be combined together to create more powerful commands).

Uses of Infinite Sequences

You can never create all the values of an infinite sequence. With a generator, you can generate as many members of a sequence of indefinite length as you like, which is useful when you do not know in advance how many values will be required. This can occur, for example, when you need to generate a value for each member of a sequence of unknown length. Such requirements can arise in many contexts—when the user is entering a series of values, when you are processing the output of another generator, and so on. (The one major advantage of sequences over generators is that you can aways find out how many elements they contain.)

This is the result of generators' "lazy evaluation"—the values are not all produced first and then consumed by the client code. Instead, when another value for the sequence is required, the generator produces it, and is then suspended (retaining the values of all local variables from the function call) until it is resumed to produce the next value in the sequence. So as long as the client code eventually stops asking for values, there really is no problem with an infinite generator. Just don't expect it to ever produce all its values—that would take an infinite amount of time!

The Itertools Module

Once generators and generator expressions were introduced into the language, iteration became a focus for development. This led to the introduction of the **itertools** module, first released with Python 2.3. itertools contains many useful functions to operate on generators and sequences. The algorithms are implemented in C, and so they run a lot faster than pure-Python equivalents. When you look at the Python documentation for the module, however, you will find that many of the functions are documented to include broadly-equivalent Python to explain them more fully.

It's important to remember that generators are a "one-shot deal": once data is consumed, it isn't possible to go back and retrieve that data again. Therefore, most of the operations you perform on generated sequences are not repeatable, unlike operations on tuples, lists, and strings.

itertools.tee: duplicating generators

tee takes two arguments: the first is a generator and the second is a count (2, if not specified). The result is

the given number of generators that can be used independently of each other.

> **Note** Because the resulting generators can be used independently, the implementation must store any values that have been consumed from one of the result generators but not from all the others. Consequently, if your code consumes most of the values from one of the result generators before the rest, you may find it more efficient to simply construct a list and use multiple iterations over that.

In your **Python4_Lesson08/src** folder, create **teesamp.py** as shown:

teesamp.py: Tee a generator to simplify program logic

```
"""
Demonstrate simple use of itertools.tee.
"""
import itertools

actions = "save", "delete"
data = ["file1.py", "file2.py", "save", "file3.py", "file4.py",
        "delete", "file5.py", "save", "file6.py",
        "file7.py", "file8.py", "file9.py", "save"]
saved = []
deleted = []

def datagen(d):
    "A 'toy' data generator using static data"
    for item in d:
        yield item

commands, files = itertools.tee(datagen(data))
for action in commands:
    if action in actions:
        for file in files:
            if file == action:
                break
            if action == "save":
                saved.append(file)
            elif action == "delete":
                deleted.append(file)
print("Saved:", ", ".join(saved))
print("Deleted:", ", ".join(deleted))
```

The program tees a single data source containing filenames and commands into two separate generators. It then iterates over the first generator until it finds a command. Then, it iterates over the second generator, performing the requested action on the files it retrieves until it "catches up" with the first generator (detected because the command is seen). This avoids the need to save the filenames in an ancillary list until the program knows what to do with them.

- Save and run it:

Results expected from teesamp.py

```
Saved: file1.py, file2.py, file5.py, file6.py, file7.py, file8.py, file9.py
Deleted: file3.py, file4.py
```

itertools.chain() and itertools.islice(): Concatenating Sequences and Slicing Generators Like Lists

The **chain()** function can be called with any number of sequences as arguments. It yields all the elements of the first sequence, followed by all the elements of the second sequence, and so on until the last sequence argument is exhausted.

It isn't possible to subscript a generator like it is a sequence such as a list or a tuple, because subscripting requires all the elements of a sequence to be in memory at the same time. Sometimes, however, you need to select elements from a generated sequence in much the same way you do for an in-memory sequence. The

itertools module allows you to do this with its **islice** function..

It takes up to four arguments: (seq, [start,] stop [, step]). If only two arguments are provided, the second argument is the length of the slice to be generated, starting at the beginning of the sequence. When three arguments are provided, the second argument M is the index of the starting element and the third argument N is the index of the element *after* the last one in the result. This closely parallels the seq[M:N] of standard sequence slicing. Finally, when all four arguments are present, the last argument is a "stride", which determines the gap between selected elements. As mentioned above, slicing operations on generated sequences will not be repeatable because the operation consumes data from the sequence, and each value can be produced only once.

The following interactive example demonstrates the use of chaining and slicing on generated sequences.

Using sequence chaining and slicing

```
>>> import itertools
>>> s1 = (1, 3, 5, 7, 11)
>>> s2 = ['one', 'two', 'three', 'four']
>>> def sqq(n):
...     for i in range(n):
...         yield i*i
...
>>> s3 = sqq(10)
>>>
>>> input = itertools.chain(s1, s2, s3)
>>> list(itertools.islice(input, 2, 7, 2))
[5, 11, 'two']
>>> list(itertools.islice(input, 3))
['three', 'four', 0]
>>>
```

It is important here to observe that the second operation on the chained sequences starts with the *first element not consumed by the previous operation*.

itertools.count(), itertools.cycle() and itertools.repeat()

These three functions provide convenient infinite seqences for use in other contexts. **count(start=0, step=1)** generates a sequence starting with the value of its **start** argument and incremented by the step amount (with a default of 1) for each call. **cycle(i)** takes an iterable argument **i** and yields each one until the sequence is exhausted, whereupon it returns to the start of the sequence and starts again. **repeat(x)** simply yields its argument **x** every time a value is requested.

itertools.dropwhile() and itertools.takewhile()

Sometimes you only want to deal with the end of a sequence, and sometimes you only want to deal with the beginning. These functions allow you to do so by providing a *predicate function* that is used to determine when to start or stop yielding elements. The function is applied to successive values in the sequence. In the case of **dropwhile()**, elements are discarded until one is found for which the function returns False, after which the remaining values are yielded without testing them. **takewhile()**, on the other hand, returns elements of the sequence until it encounters one for which the function returns False, at which point it immediately raises a StopIteration exception.

You can learn a little more about these functions in an interactive console session.

Experimenting with dropwhile() and takewhile()

```
>>> import itertools
>>> def lt5(n):
...     return n<5
...
>>> s1 = [1, 3, 2, 4, 6, 4, 2, 3, 1]
>>> list(itertools.dropwhile(lt5, s1))
[6, 4, 2, 3, 1]
>>> list(itertools.takewhile(lt5, s1))
[1, 3, 2, 4]
>>>
```

For any function **f** and sequence **s**:

list(takewhile(f, s)) + list(dropwhile(f, s)) == list(s)

The two functions are therefore complementary in nature.

This has "scratched the surface" of the itertools module, but there is plenty more to reward your reading of its documentation should you feel so inclined.

Generator Expressions

In the same way that list comprehensions offer a more succinct way to create lists, generator expressions help you to use generators without having to write a generator function.

Since list and tuple creation is relatively fast in Python, you will probably find that you have to be working with fairly large data sets in order to see compelling advantages for generators over lists. Try it with some sample random data to get a feel for the relative speed of lists. In this example, we'll sum a bunch of numbers from a list of random numbers between 0 and 1 in two ways: the first sums the values using a generator expression, the second creates a list and sums that.

Note

The lists get so large it is entirely possible that there is not enough memory to create the larger ones. In that case, you may see **MemoryError** exceptions such as the one demonstrated below (this particular run was made on a testing machine with limits on the amount of memory one process can use, so you may not see the exception because you are using better-resourced production machines in your lab sessions). When you are finished with the interactive session, you should terminate the console with the  button and start a new console session. Once an interpreter process has suffered a memory error, it may not be able to reclaim all that memory, so it is best to start again.

Test the relative speed of lists and generator expressions

```
>>> from random import random
>>> from timeit import timeit
>>> for i in (10000, 100000, 1000000, 10000000, 20000000, 50000000):
...     lst = [random() for j in range(i)]
...     print("Length", i)
...     print(timeit("sum(x+1 for x in lst)", "from __main__ import lst", number=1))
...     print(timeit("sum([x+1 for x in lst])", "from __main__ import lst", number=1))
...
Length 10000
0.0032087877090524073
0.0031928638975065268
Length 100000
0.032067762802255595
0.03326847406583798
Length 1000000
0.18962521773018637
0.2972891806081499
Length 10000000
2.405814984865395
2.7992426411736684
Length 20000000
4.417569830802519
5.341360144934622
Length 50000000
10.820288612100143
Traceback (most recent call last):
  File "<console>", line 5, in <module>
  File "C:\Python\lib\timeit.py", line 213, in timeit
    return Timer(stmt, setup, timer).timeit(number)
  File "C:\Python\lib\timeit.py", line 178, in timeit
    timing = self.inner(it, self.timer)
  File "<timeit-src>", line 6, in inner
  File "<timeit-src>", line 6, in <listcomp>
MemoryError
>>>
```

The code uses **timeit**'s **number** argument to ensure that only one timed operation of the sample code is run. This means that the timings are not necessary repeatable, but are at least indicative of the relative times of the different operations. It seems that, the longer the sequence, the more improvement you can expect to see from using a generator expression. For comparison with the timings on Windows, here is the output from a MacOS machine (with more memory) running the same code in a new Python console session.

Results of the same test on a different machine

```
Length 10000
0.00169491767883
0.00142598152161
Length 100000
0.017655134201
0.0198609828949
Length 1000000
0.18835401535
0.206699848175
Length 10000000
1.77904486656
2.16294407845
Length 20000000
3.62438511848
4.16168618202
Length 50000000
9.03414511681
76.5883550644
>>>
```

You can see that there is sufficient memory for this computer to create the larger lists. While the performance of the list-based technique and the generator expressions are the same, the difference does not seem to be quite as marked. These tests were run on a different operating system, which may have something to do with it. Note that with fifty million elements in the last test iteration, the creation of the list starts to add large overhead, and the generator expression is markedly faster.

You have already come across list comprehensions such as **[x*x for x in sequence]**. You can, if you want, think of list comprehensions as generator expressions surrounded by list brackets. The brackets tell the interpreter that it is required to create a list, so it runs the generator to exhaustion and adds each element to a newly-created list. There is no essential difference between the expression above and **list(x*x for x in sequence)**, but the latter does seem to be about 25% slower on implementations current at the time of writing, whether the sequence is a list or a generator function.

Generators, while a relatively late addition to the Python language, are rapidly becoming an essential part of it. When you are dealing with large data sets, a good command of generators can make all the difference between a slow program and a fast one. It is therefore important to be aware of their possibilities. This is not too difficult, once you realise that they are often simply a faster and more efficient way to handle data.

When you finish the lesson, don't forget to complete the homework!

Uses of Introspection

Lesson Objectives

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

- explain 'Introspection.'
- attribute Handling Functions.
- use Introspection.
- use the Inspect Module.

The Meaning of 'Introspection'

The word "introspection" means "looking inside." Introspective people are ones who think about themselves, usually to increase self-understanding. In Python, introspection is a way that your programs can learn about the environment in which they operate and the properties of the modules they import.

You have already learned about several of Python's introspection mechanisms. The built-in dir() function, for example, attempts to return (to quote from the documentation) "an interesting set of names"—meaning the names of attributes accessible from the object passed as an argument. If no argument is passed, it returns the attributes found in the current local namespace.

dir() in Python 3.x has a hook that looks for a __dir__() method on its argument. If such a method is present, it is called and dir() returns what the method returns. This allows you to determine what users see about your object, and this can be useful if you are using "virtual" attributes (that is, if your objects handle access to methods that do not appear in the class's __dict__). If no __dir__() method is found, dir() uses a standard mechanism to compose its result after examining its argument.

Some Simple Introspection Examples

x.__class__.__name__ will tell you the name of an object's class (and is much more reliable than trying to analyze a repr() string):

The Right and Wrong Way to Extract a Class Name

```
>>> class Something:
...     pass
...
>>> s = Something()
>>> s
<__main__.Something object at 0x10063be50>
>>> repr(s)[1:-1].split()[0].split(".")[1] # WRONG!
'Something'
>>> s.__class__.__name__ # RIGHT (AND SO MUCH EASIER)
'Something'
>>> repr(4)[1:-1].split()[0].split(".")[1] # Fail
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: list index out of range
>>> 4.__class__.__name__
  File "<console>", line 1
    4.__class__.__name__
             ^
SyntaxError: invalid syntax
>>> (4).__class__.__name__ # SUCCEED
'int'
>>> str(type(4))[1:-1].split()[1][1:-1] # Way too complex
'int'
>>> str(type(s))[1:-1].split()[1][1:-1] # And only give same result for built-in
s (see s above)
'__main__.Something'
```

The failed attempt to extract the class name from the integer 4's repr() string shows just how fragile the "wrong" method is: it applies only to objects with a very specific representation. When handed an int instance it explodes, raising an exception. The syntax error occurred because the interpreter took the period (".") to be part of a number, and then could not understand why it was followed by an identifier. Putting the (4) in parentheses allows the lexical analysis routines to parse things correctly, and we see that the class name is available from built-in classes just as it is on self-declared ones. If you find yourself writing code like the first and last examples, you should question whether there isn't a better way: Python is designed to avoid the need for such contortions.

some_object.__doc__ can be useful, but if things are properly written, you'll get better presentation from **help(some_object)**, which is designed to print necessary documentation in a legible way.

Attribute Handling Functions

If you took earlier courses in this Certificate Series (or otherwise possibly from private study) you've encountered the getattr(**obj**), setattr(**obj**), and delattr(**obj**) functions, and learned that they result in a call to their argument **obj**'s __getattr__(), __setattr__(), and __delattr__() methods. There is also the hasattr() predicate, which can be used to determine whether or not a given attribute is present in an object. There is, however, no corresponding __hasattr__() method. You might wonder what hasattr() does to find out what value to return, and the answer to that question is complex enough to have received the attention of some of the best minds in Python.

Without going too deeply into the internals, it is fairly easy for you to determine whether or not __getattr__() gets called by hasattr() under at least some circumstances. You simply write a class whose instances report calls of their __getattr__() method, and then call hasattr() on an instance:

INTERACTIVE SESSION:

```
>>> class X:
...     def __getattr__(self, name):
...         print("getattr", name)
...         return 0
...
>>> x = X()
>>> hasattr(x, "thing")
getattr thing
True
>>>
```

hasattr(obj, "__call__") can be used to tell you whether or not **obj** can be called like a function. Older versions of Python provide a callable() built-in function, which should have been removed in Python 3.0 because the given test is now all that is required—everything callable has a __call__ attribute. Its deletion was omitted in error for the 3.0 release, with the result that callable *is* available for that release. It was then removed from 3.1 (the version in use when this course was being written), but has returned in 3.2 because the above test turns out not to be quite as specific as the version that can be written in C with full access to the object structures. Being able to determine the presence or absence of a particular attribute is occasionally useful in other contexts.

Note You should avoid writing code where "too much" (a judgment call) of the logic depends on the presence or absence of specific attributes, unless you are writing deliberately introspective code as part of a framework or library.

Of course you can implement whole "virtual namespaces" within your own objects by using getattr() and setattr(), but remember that these functions can also be used (assuming you can gain access to the required namespaces) to modify your current environment. Understand that doing so in this way is not recommended except in rather extreme cases, because it results in "magical" changes—changes whose origin is difficult or impossible to discern by reading the program code:

'Magical' changes to the module's namespace

```
>>> import sys
>>> __name__
'__main__'
>>> module = sys.modules[__name__]
>>> a
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'a' is not defined
>>> setattr(module, "a", 42)
>>> a
42
>>>
```

Before the setattr() call, there was no "a" defined in the module's namespace. Since all imported modules are available under their natural names from **sys.modules**, you can access the current module's namespace by looking it up.

If it were possible to subclass the module object to change its attribute access methods, we could be faced with some extremely hard-to-understand code! Fortunately this is not something you need to worry about in practice. Most of the code you will encounter does not use such tricks (indeed, the Django framework mentioned earlier had a period in its development devoted to "magic removal" to make the code easier for Python programmers and beginners to understand, and provide a framework that was less brittle).

What Use is Introspection?

Frameworks use introspection frequently, to discover the capabilities of objects the user has passed; for example, "does this object's class have a **something()** method? If so, call the object's **do_something()**; otherwise call the **do_something_similar()** framework function with the object as an argument." Some built-in functions also do this kind of introspection. The dir() built-in mentioned above returns the result of the argument object's __dir__() method if it has one; otherwise it uses built-in functionality to provide an "interesting" set of names (the result is not defined more clearly than that anywhere in the code).

Note A *framework* is an environment that provides a wealth of facilities to programmers. You can think of it as being like an "operating system for a particular type of programming task." The users of frameworks are generally application programmers, using the framework (for example, Django or Tkinter) to build a particular type of application (in Django's case, they would be web applications; in Tkinter's case, they would be windowed applications).

The Inspect module

This module allows you to dig as deep as you ever need to in terms of introspection. It provides many functions by which you can determine the properties of objects, incuding sixteen predicates that allow you to easily determine whether an object is of a particular type.

The getmembers() Function

inspect.getmembers(obj[, predicate]) returns a list of two-element (name, value) tuples. If you provide a second argument, it is called with the value as its only argument and the item only appears in the resulting list if the result is True. This makes the predicates mentioned in the last paragraph very useful if you are only interested in objects of a particular type. Following are some special attributes especially worth knowing about (columns to the right explain which attributes you can expect to see on five given types of object).

Attribute	Purpose	Module	Class	Method	Function	Built-in
__doc__	Documentation string	✓	✓	✓	✓	✓
__file__	Path to the file from which the object was loaded	✓				
__module__	Name of module in which the object was imported		✓		✓	

__name__	Name of object			✓	✓	✓
__func__	The implementation of the method			✓		
__self__	Instance to which this method is bound (or **None**)			✓		✓
__code__	Code object containing function's bytecode				✓	
__defaults__	Documentation string	✓	✓	✓	✓	
__globals__	Documentation string	✓	✓	✓	✓	

The predicates that you can use with **getmember()** are:

Predicate name	Purpose
ismodule(x)	Returns True if x is a module.
isclass(x)	Returns True if x is a class, whether built-in or user-defined.
ismethod(x)	Returns True if x is a bound method written in Python.
isfunction(x)	Returns True if x is a function (including functions created by lambda expressions).
isgeneratorfunction(x)	Returns True if x is a Python generator function.
isgenerator(x)	Returns True if x is a generator.
istraceback(x)	Returns True if x is a *traceback object* (created when an exception is handled).
isframe(x)	Returns True if x is a *stack frame* (can be used to debug code interactively).
iscode(x)	Returns True if x is a *code object*.
isbuiltin(x)	Returns True if x is a built-in function or a bound built-in method.
isroutine(x)	Returns True if x is a user-defined or built-in function or method.
isabstract(x)	Returns True if x is an *abstract base class* (one meant to be inherited from rather than instantiated).
ismethoddescriptor(x)	Returns True if x is a method descriptor *unless* ismethod(x), isclass(x), isfunction(x) or isbuiltin(x) is True.
isdatadescriptor(x)	Returns True if x is a *data descriptor* (has both a **__get__()** and a **__set__()** method).
isgetsetdescriptor(x)	Returns True if x is a getsetdescriptor—these are used in extension modules.
ismemberdescriptor(x)	Returns True if x is a member descriptor—these are used in extension modules.

The second argument to inspect.getmembers() allows you to access members of a particular type easily:

Experimenting with getmembers()

```
>>> import inspect
>>> from smtplib import SMTP
>>> from pprint import pprint
>>> pprint(inspect.getmembers(SMTP))
[('__class__', <class 'type'>),
 ('__delattr__', <slot wrapper '__delattr__' of 'object' objects>),
 ('__dict__', <dict_proxy object at 0x1006b7910>),
 ('__doc__',
  "This class manages a connection to an SMTP or ESMTP server.\n
    SMTP Objects:\n
        SMTP objects have the following attributes:\n
            helo_resp\n
                This is the message given by the server in response to the\n
                most recent HELO command.\n\n
            ehlo_resp\n
                This is the message given by the server in response to the\n
                most recent EHLO command. This is usually multiline.\n\n
            does_esmtp\n
                This is a True value _after you do an EHLO command_, if the\n
                server supports ESMTP.\n\n
            esmtp_features\n
                This is a dictionary, which, if the server supports ESMTP,\n
                will _after you do an EHLO command_, contain the names of the\n
                SMTP service extensions this server supports, and their\n
                parameters (if any).\n\n
                Note, all extension names are mapped to lower case in the\n
                dictionary.\n\n
        See each method's docstrings for details.  In general, there is a\n
        method of the same name to perform each SMTP command.  There is also a\n
        method called 'sendmail' that will do an entire mail transaction.\n
   "),
 ('__eq__', <slot wrapper '__eq__' of 'object' objects>),
 ('__format__', <method '__format__' of 'object' objects>),
    ...
 ('__str__', <slot wrapper '__str__' of 'object' objects>),
 ('__subclasshook__',
  <built-in method __subclasshook__ of type object at 0x1b002aed0>),
 ('__weakref__', <attribute '__weakref__' of 'SMTP' objects>),
 ('_get_socket', <function _get_socket at 0x1007a3d98>),
 ('close', <function close at 0x116437958>),
    ...
 ('verify', <function verify at 0x116437628>),
 ('vrfy', <function verify at 0x116437628>)]
>>>
>>> pprint(inspect.getmembers(SMTP, inspect.ismethod))
[]
>>> pprint(inspect.getmembers(SMTP, inspect.isfunction))
[('__init__', <function __init__ at 0x1007a3c88>),
 ('_get_socket', <function _get_socket at 0x1007a3d98>),
 ('close', <function close at 0x116437958>),
    ...
 ('verify', <function verify at 0x116437628>),
 ('vrfy', <function verify at 0x116437628>)]
>>> smtp = SMTP()
>>> pprint(inspect.getmembers(smtp, inspect.ismethod))
[('__init__',
  <bound method SMTP.__init__ of <smtplib.SMTP object at 0x100644c90>>),
 ('_get_socket',
  <bound method SMTP._get_socket of <smtplib.SMTP object at 0x100644c90>>),
 ('close', <bound method SMTP.close of <smtplib.SMTP object at 0x100644c90>>),
    ...
 ('verify',
  <bound method SMTP.verify of <smtplib.SMTP object at 0x100644c90>>),
 ('vrfy', <bound method SMTP.verify of <smtplib.SMTP object at 0x100644c90>>)]
>>>
```

You will get rather more output than we showed here, and the docstring has been reformatted to make it easier to read in the listing, but there is no reason to list eveything that is output. The detail presented is sufficient to demonstrate that the SMTP class has many member attributes, including the standard "dunder" names, many of them inherited from the **object** type.

Asking for the methods of the class (using the **ismethod()** predicate as a second argument to **getmembers()**) changes it to return the empty list. This is not too surprising, as the predicate is documented as returning True only for *bound* methods—methods associated with a particular instance. The **isfunction()** predicate used in the third example returns the methods that are specifically defined on the class, but not those inherited from superclasses (which in practice means the **object** type). Creating an instance of the SMTP class and querying that for methods gives a much more interesting result.

Introspecting Functions

There are various attributes of a code object that can be used to discover information about the function to which it belongs. The **inspect** module provides some convenience functions to avoid the need to use them under most circumstances, however.

inspect.getfullargspec(f) returns a named tuple **FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)** containing information pertaining to the function argument f:

- **args** is a list of the names of the standard (positional *and* keyword) arguments.
- The **defaults** member contains the default values for the arguments specified by keyword (which always follow the positionals).
- **varargs** and **varkw** are the names of the * and ** arguments, if present. The value **None** is used when there are no such arguments.
- **kwonlyargs** is a list of the arguments that *must* be provided as keyword arguments
- **kwonlydefaults** is the list of default values of those arguments.
- **annotations** is a dict that maps argument names to annotations (which will usually be empty, because we will not cover function annotations in this course)

inspect.formatargspec(args[, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations]) takes the output from **getfullargspec()** and re-creates the arguments part of the function signature.

Here is a little example to show you how they work.

Function introspection

```
>>> import inspect
>>> def f(a, b, c=1, d="one", *args, **kw):
...     print('a', a, 'b', b, 'c', c, 'd', d, 'args', args, 'kw', kw)
...
>>> inspect.getfullargspec(f)
FullArgSpec(args=['a', 'b', 'c', 'd'], varargs='args', varkw='kw', defaults=(1,
'one'), kwonlyargs=[], kwonlydefaults=None, annotations={})
>>> inspect.formatargspec(*inspect.getfullargspec(f))
"(a, b, c=1, d='one', *args, **kw)"
>>>
```

As you can see, **formatargspec()** produces a parenthesized list of argument specifications that can easily be translated back into the original format (or something equivalent to it) using the **formatargspec()** function.

There are other facilities that come as part of the **inspect** module, and you can read the documentation for that module when you feel the need to learn more. Using the features you have learned about in this lesson, however, you should be able to discover what your program needs to know about the code that surrounds it.

When you finish the lesson, don't forget to complete the homework!

Multi-Threading

Lesson Objectives

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

- utilize Threads and Processes.
- use the Threading Library Module.

Threads and Processes

When you are new to programming (as some students were when they started this Certificate Series), you don't necessarily think too much about all the other things that the computer is doing besides running your programs. You connect to a Windows system using remote desktop protocols, and the same computer that is supporting your session may be supporting other student sessions as well. It has to share its attention between these different tasks, as well as handling your keyboard and mouse input and providing output in various GUIs. There is an enormous amount of activity going on in a modern server computer.

Multiprogramming

Early computers worked on exactly one problem at a time. As their resources grew and they became faster, people observed that much of the computer's time was spent idling, waiting for some external event (such as reading an 80-column card punched with data). Techniques were developed to allow several programs to reside in the computer at the same time, so that when one program was waiting, the processor could be working on another. The classic name for this technique is *multiprogramming*.

In a modern computer, each program is written as though it had exclusive use of the machine that it runs on, even though in fact the operating system will share its available processing power among hundreds or even thousands of *processes*. Each process is isolated from the others by running in a special protected mode, which can only access the memory that the operating system has allocated to it. To use storage and communications features, for example, processes have to make calls to the operating system. Thus the separate processes are isolated from each other. Only the operating system has the ability to access all processes' memory.

Multiprocessing

Nowadays, the engineers who design the chips that go into computers are running up against some fairly fundamental speed constraints. Generally you can make things run faster by making them smaller (because this reduces the travelling time of the minute almost-light-speed electrical currents on which logic circuits rely). The faster a circuit works, the more energy it dissipates as heat. But when you make the chips too small or too fast they melt, because too much energy is being dissipated in too small a space, leading to overheating.

To try and overcome the speed limitations chip designers have started instead to build computers with more than one processor on the same chip, and computer engineers are putting several of those chips on a single motherboard to build so-called multi-processor computers. The different processors share memory and peripherals but are otherwise independent of each other. As long as there are no conflicting requirements for resources, each of the processors can be running a different process in parallel—literally, the different processes are executed at the same time on different processors, and the operating system tries to keep all the processors as busy as it can. So speed increases today are being achieved by running several computations in parallel on separate processors. This ability to execute several instruction streams truly simultaneously is referred to as *multiprocessing*.

Multi-Threading

In the same way that the operating system shares the processor power between lots of processes all contending for its use at certain times, so you can write programs that take a similar approach. They manage lots of separate activities in essentially the same way, but independently of each other. Each independent activity is usually referred to as a *thread*, and programs that manage multiple threads are said to be *multi-threaded*.

For example, around the turn of the century I was asked to help a client send its monthly invoices out by e-mail. It was impractical to write a program that sent the emails one by one. Firstly, formulating the messages took a significant amount of time, with waits for data to come in from the database and the networked domain

name system that translates names like holdenweb.com into IP addresses like 174.120.139.138. Furthermore, there can be significant holdups in communication when a server is no longer present, and a connection attempt takes minutes to time out. Early experiment established that it would take upwards of two days to send out the invoices, and that performance would be flaky with occasional complete hang-ups.

Consequently, I had to take a different approach. Because I had written the code to send an email as a Python function, it was relatively easy to refactor the code so that the function became the **run()** method of a Python **threading.thread** subclass. This allowed me to easily create threads to send individual emails. Some additional plumbing was required, with a thread extracting invoicing tasks from the database, dispatching threads to send the emails, and finally updating the database with the record of success or failure. The plumbing code could easily be adjusted to create and use any number of threads, and after a very short time the client was able to send out almost 50,000 emails in under two hours using 200 parallel threads.

That represented a monthly saving of at least $10,000 to the client in postage, so the time spent programming was well worthwhile.

Threading, Multiprocessing, CPython and the GIL

The CPython implementation of Python is currently the only implementation of Python 3, though the developers of the other major implementations (PyPy, Jython and IronPython) have all expressed a commitment to support this latest version of Python. The CPython implementation retains a feature from Python version 2 (which was the basis for development of the Python 3.x code)—the so-called *Global Interpreter Lock*, better known as the *GIL*.

Only one thread in a Python program can hold the GIL at any time. In effect this means that *multi-threaded programs in Python find it very difficult to take advantage of more than one processor*—the purpose of the GIL is to allow speed-up of common primitive operations by ensuring that the same object is never being accessed in incompatible ways at the same time by two processors.

Guido van Rossum, Python's inventor, is on public record as saying that he sees no reason to remove the GIL from CPython. He suggests that people wanting to take advantage of hardware parallelism should either write their applications to run as multiple cooperating processes or use a Python implementation that does not rely on a GIL for thread safety. As you will see in a later lesson, once you understand how to use the **threading** library, it is not much more effort to use the **multiprocessing** library to achieve a true multi-process solution. Since this runs multiple processes rather than multiple threads, each process runs with an independent interpreter, and can take full advantage of multiprocessing hardware if processes are created in sufficient number.

In essence you will only see benefits from multi-threading if the tasks performed by each thread require significant "waiting time" (such as awaiting a response from a user, or from a remote computer, or from some file). In CPython only one thread at a time can hold the GIL, so multiple threads can only take advantage of multiple processors if they use C extensions specifically written to release the GIL while performing work that does not require access to the interpreter's resources. Multi-threaded solutions are frequently seen as "difficult" to communicate to beginners, but most threading problems seem to come from not retaining strict isolation between the namespaces and object space used by different threads. This is not as simple as it seems, because some standard library functions can alter the environment of *all threads in a particular process*.

The Threading Library Module

threading is the primary library for handling threads in Python. In many implementations, you will find there is also an underlying **_thread** module, used to access threading libraries from the underlying system. In all cases, the threading library works in roughly the same way.

When multiple threads are present, the interpreter will share its time between the threads. Threads can become blocked for the same reasons that processes can become blocked: they need to wait for something (incoming network data, a connection request, data from filestore). In CPython, the interpreter runs a certain number of bytecodes of one thread before moving on to the next in a round-robin between non-blocked threads. If a thread is holding the GIL, no other threads can be scheduled (except those that have explicitly released it, usually in an extension module).

Creating Threads (1)

The simplest way to create a new thread is by instantiating the **threading.thread** class. You are expected to provide a **target** keyword argument, which will be called in the context of the new thread when it is started. You can also provide **args**, a tuple of positional arguments and **kwargs**, a dict of keyword arguments. These arguments will be passed to the **target** call when the thread is started. Finally, you can give your thread a name if you want by passing a **name** keyword argument. Default names for threads are typically names like "Thread-N." Create a **Python4_Lesson10** project and assign it to the **Python4_Lessons** working set, and then, in your **Python4_Lesson10/src** folder, create **thread.py** as shown:

thread.py: doing six things in parallel

```
"""
thread.py: demonstrate creation and parallel execution of threads.
"""

import threading
import time

def run(i, name):
    """Sleep for a given number of seconds, report and terminate."""
    time.sleep(i)
    print(name, "finished after", i, "seconds")

for i in range(6):
    t = threading.Thread(target=run, args=(i, "T"+str(i)))
    t.start()
print("Threads started")
```

The program defines a function that sleeps for a while, then prints a message and terminates. It then loops, creating and starting six threads, each of which uses the function to sleep a second longer than the last before reporting, using its given name. When you run this program, you see:

Results of running thread.py

```
T0 finished after 0 seconds
Threads started
T1 finished after 1 seconds
T2 finished after 2 seconds
T3 finished after 3 seconds
T4 finished after 4 seconds
T5 finished after 5 seconds
```

As soon as the interpreter has more than one active thread it starts sharing its time between the threads. This, coupled with the zero wait time for the first task, means that the very first thread created has finished even before the main thread has completed its creation and starting of all six threads (which is when it prints the "Threads started" message. The other threads then report in at one-second intervals.

Note When a running program is associated with the console window, its "Terminate" and "Terminate All" icons will be red, indicating that the console is monitoring an active process. As you run the program, you will see that even though the main thread (the one which started program execution) terminates, Ellipse still shows the console as containing an active process until the last thread has terminated.

When Python creates a new thread, that thread is to a degree isolated from the other threads in the same process. Threads can share access to module-global variables, although you must be very careful not to change anything that could be changed concurrently by any other thread. There are safe ways for threads to communicate with each other (discussed in the next lesson), and you should use those. The namespace of the function call that starts the thread is unique to the thread, however, and any functions that are called similarly have new namespaces created.

Waiting for Threads

Our initial thread.py program just assumed that the threads would all terminate in the end and everything would come out nicely. If you don't want to make this assumption, you can either monitor the thread count or you can wait for individual threads. The first approach is rather simpler, but it relies on your main thread being the only part of the program that is creating threads. Otherwise, the thread count would vary apparently randomly. The function to access the current number of threads is **threading.active_count()**.

Modify thread.py to monitor the number of active threads

```
"""
thread.py: demonstrate simple monitoring of execution of threads.
"""

import threading
import time

def run(i, name):
    """Sleep for a given number of seconds, report and terminate."""
    time.sleep(i)
    print(name, "finished after", i, "seconds")

bgthreads = threading.active_count()
for i in range(6):
    t = threading.Thread(target=run, args=(i, "Thread-"+str(i)))
    t.start()
print("Threads started")
while threading.active_count() > bgthreads:
    print("Tick ...")
    time.sleep(2)
print("All threads done")
```

- Your output looks like this:

Output of updated thread.py

```
Thread-0 finished after 0 seconds
Threads started
Tick ...
Thread-1 finished after 1 seconds
Thread-2 finished after 2 seconds
Tick ...
Thread-3 finished after 3 seconds
Thread-4 finished after 4 seconds
Tick ...
Thread-5 finished after 5 seconds
All threads done
```

The program now takes a thread count before starting any threads, and then after starting them waits in a timed loop until the thread count returns to what it was before. An alternative is to wait for each thread to complete by calling its **join()** method. This blocks the current thread until the thread whose **join()** method was called has finished. Generally this works best when the order of the threads is known, or unimportant: once your thread blocks on a join() it can do *nothing* until that thread terminates.

Modify thread.py to wait for each thread using join()

```
"""
thread.py: demonstrate thread monitoring by awaiting termination.
"""

import threading
import time

def run(i, name):
    """Sleep for a given number of seconds, report and terminate."""
    time.sleep(i)
    print(name, "finished after", i, "seconds")

bgthreads = threading.active_count()
threads = []
for i in range(6):
    t = threading.Thread(target=run, args=(i, "Thread-"+str(i)))
    t.start()
    threads.append((i, t))
print("Threads started")
while threading.active_count() > bgthreads:
    print("Tick ...")
    time.sleep(2)
for i, t in threads:
    t.join()
    print("Thread", i, "done")
print("All threads done")
```

The "worker" threads actually terminate in the order in which the main thread created—and waits for—them, and so the output shows each thread logged as terminated as soon as it terminates.

Threads finish in the same order the main thread waits

```
Thread-0 finished after 0 seconds
Threads started
Thread 0 done
Thread-1 finished after 1 seconds
Thread 1 done
Thread-2 finished after 2 seconds
Thread 2 done
Thread-3 finished after 3 seconds
Thread 3 done
Thread-4 finished after 4 seconds
Thread 4 done
Thread-5 finished after 5 seconds
Thread 5 done
All threads done
```

A very simple modification to the source makes the threads started earlier finish later:

thread.py still waits, but worker threads finish last first

```
"""
thread.py: demonstrate thread monitoring by awaiting termination.
"""

import threading
import time

def run(i, name):
    time.sleep(i)
    print(name, "finished after", i, "seconds")

threads = []
for i in range(6):
    t = threading.Thread(target=run, args=(6-i, "Thread-"+str(i) ))
    t.start()
    threads.append((i, t))
print("Threads started")
for i, t in threads:
    t.join()
    print("Thread", i, "done")
print("All threads done")
```

This time the threads are all reported together, because by the time the first thread completes, all others have already completed, and so their **join()** methods return immediately. This changes the nature of the output somewhat.

Once the first join() returns so will all others

```
Threads started
Thread-5 finished after 1 seconds
Thread-4 finished after 2 seconds
Thread-3 finished after 3 seconds
Thread-2 finished after 4 seconds
Thread-1 finished after 5 seconds
Thread-0 finished after 6 seconds
Thread 0 done
Thread 1 done
Thread 2 done
Thread 3 done
Thread 4 done
Thread 5 done
All threads done
```

Creating Threads (2)

The second way to create threads is to define a subclass of **threading.thread**, overriding its run() method with the code you want to run in the threaded context. In this case, you are expected to pass any data in through the __init__() method, which also means making an explicit call to **threading.Thread.__init__()** with appropriate arguments. So there is a cost associated with creating threads this way, because the programming is a little more detailed.

The approach can win if the logic gets complex, however, because other methods can be added to the subclass and used to implement complex functionality in a reasonably modular way: all logic is still attached to a single class. Further, each thread is a separate instance of the class and so the methods can communicate via instance variables as well as explicit arguments. When the thread is run as a function, there is no corresponding "global" namespace that can be used.

First let's try and re-cast the thread.py program to use a threading.thread subclass. When you use such subclasses, it is possible to access the thread name, so the only argument required will be the sleep time. This argument is saved in an instance variable, and any other arguments are passed to the standard thread initialization routine (though arguments are not normally passed to instantiate subclasses with run() methods, who knows how the API may change in the future—this way is future-proof). When the thread is started, its run() method begins to execute and the sleep time is extracted from the instance variable. As before, the main thread ticks every two seconds and waits for the thread count to go back to its "main thread only" value.

Modify thread.py to subclass threading.thread

```
"""
thread.py: Use threading.Thread subclass to specify thread logic in run() method
.
"""
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, sleeptime, *args, **kw):
        threading.Thread.__init__(self, *args, **kw)
        self.sleeptime = sleeptime
    def run(self):
        print(self.name, "started")
        time.sleep(self.sleeptime)
        print(self.name, "finished after", self.sleeptime, "seconds")

def run(i, name):
    time.sleep(i)
    print(name, "finished after", i, "seconds")

threads = []
bgthreads = threading.active_count()
tt = [MyThread(i+1) for i in range(6)]
for t in tt:
for i in range(6):
    t = threading.Thread(target=run, args=(6-i, "Thread "+str(i) ))
    t.start()
    threads.append((i, t))
print("Threads started")
for i, t in threads:
    t.join()
    print("Thread", i, "done")
while threading.active_count() > bgthreads:
    time.sleep(2)
    print("tick")
print("All threads done")
```

There should be no surprises in the output:

Subclassing threading.thread works too!

```
Thread-1 started
Thread-2 started
Thread-3 started
Thread-4 started
Thread-5 started
Thread-6 started
Threads started
Thread-1 finished after 1 seconds
tick
Thread-2 finished after 2 seconds
Thread-3 finished after 3 seconds
Thread-4 finished after 4 seconds
tick
Thread-5 finished after 5 seconds
Thread-6 finished after 6 seconds
tick
All threads done
```

So far, the threads we've written haven't done very much—simply sleeping and printing a message doesn't really amount to a convincing computation. The computer is still doing nothing but wait (in our process) for sleep times to expire. Now let's see what happens when we replace the sleep with some real computation.

Modifying thread.py to compute instead of sleep

```
"""
thread.py: Use threading.Thread subclass to specify thread logic in run() method
.
"""
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, sleeptime, *args, **kw):
        threading.Thread.__init__(self, *args, **kw)
        self.sleeptime = sleeptime
    def run(self):
        print(self.name, "started")
        time.sleep(self.sleeptime)
        for i in range(self.sleeptime):
            for j in range(500000):
                k = j*j
            print(self.name, "finished pass", i)
        print(self.name, "finished after", self.sleeptime, "seconds")

bgthreads = threading.active_count()
tt = [MyThread(i+1) for i in range(6)]
for t in tt:
    t.start()
print("Threads started")
while threading.active_count() > bgthreads:
    time.sleep(2)
    print("tick")
print("All threads done")
```

You can see that this time the output from the different threads is intermingled, indicating that all active threads are receiving some processor time rather than one thread running until it finishes. Without this "scheduling" behavior, threading would not be very popular.

thread.py now shows threads sharing compute resource

```
Threads started
Thread-1 finished pass 0
Thread-1 finished after 1 seconds
Thread-4 finished pass 0
Thread-3 finished pass 0
Thread-2 finished pass 0
Thread-5 finished pass 0
Thread-6 finished pass 0
Thread-4 finished pass 1
Thread-3 finished pass 1
Thread-2 finished pass 1
Thread-2 finished after 2 seconds
Thread-5 finished pass 1
Thread-4 finished pass 2
Thread-5 finished pass 2
Thread-6 finished pass 1
Thread-3 finished pass 2
Thread-3 finished after 3 seconds
Thread-4 finished pass 3
Thread-4 finished after 4 seconds
Thread-6 finished pass 2
Thread-5 finished pass 3
tick
Thread-6 finished pass 3
Thread-6 finished pass 4
Thread-5 finished pass 4
Thread-5 finished after 5 seconds
Thread-6 finished pass 5
Thread-6 finished after 6 seconds
tick
All threads done
```

Your results will probably differ from those shown above, precisely because the way the different threads are scheduled may well not be as "equitable" as you think. When you look at the long-lived threads, you can see that Thread-4 finishes pass 3 before Thread-6 has finished pass 2. But ultimately all threads are computing and they are all "pushed along" at roughly the same speed.

Multi-threading is one way to achieve asynchronous processing. For the CPython implementation (and others relying on single-processor guarantees to speed processing) this will not help if the application is CPU-bound, as all processing must take place on a single processor, and so the application cannot benefit from multiple processors in the computer it runs on.

Next, we will consider how to synchronize multiple threads, amd how to pass data safely from one thread to another.

When you finish the lesson, don't forget to complete the homework!

More on Multi-Threading

Lesson Objectives

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

- synchronize threads.
- access the Queue Standard Library.

Thread Synchronization

threading.Lock Objects

Because attempts to access (and particularly to modify) the same resource from different threads can be disastrous, the threading library includes **Lock** objects that allow you to place a *lock* on resources, stopping any other thread that tries to access the resource in its tracks (in fact, stopping any thread that attempts to acquire the same lock). A **threading.Lock** has two states: locked and unlocked, and it is created in the unlocked state.

When a thread wants to access the resource associated with a specific Lock, it calls that Lock's acquire() method. If the Lock is currently locked, the acquiring thread is blocked until the Lock becomes unlocked and allows acquisition. If the Lock is unlocked, it is locked and acquired immediately. A Lock object becomes unlocked when its release() method is called.

In the next example, we'll modify the thread.py code from the last lesson so that the "critical resource" is the ability to sleep. Before sleeping for a tenth of a second each thread has to acquire a single lock shared between all threads. Even though each thread only has to sleep for a total of a second, because there are six threads and only one of them can be sleeping at a time, it takes the program six seconds to run.

Modify thread.py to lock while sleeping

```
"""
thread.py: Use threading.Lock to ensure threads run sequentially.
"""
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, lock, *args, **kw):
        threading.Thread.__init__(self, *args, **kw)
        self.sleeptime = sleeptime
        self.lock = lock
    def run(self):
        for i in range(10):
            for j in range(500000):
                k = j*j
            self.lock.acquire()
            time.sleep(0.1)
            self.lock.release()
            print(self.name, "finished pass", i)
        print(self.name, "finished")
        print(self.name, "finished after", self.sleeptime, "seconds")

lock = threading.Lock()
bgthreads = threading.active_count()
tt = [MyThread(lock) for i in range(6)]
for t in tt:
    t.start()
print("Threads started")
while threading.active_count() > bgthreads:
    time.sleep(2)
    print("tick")
print("All threads done")
```

Save and run it:

The threads appear to finish deterministically in Eclipse

```
Threads started
tick
tick
tick
Thread-1 finished
Thread-2 finished
Thread-3 finished
Thread-4 finished
Thread-5 finished
Thread-6 finished
tick
All threads done
```

In different environments, however, the output from this program will typically vary each time you run it, because there are enough acquisitions and releases to allow different threads to get an advantage in the scheduling (which is not a simple deterministic round-robin). Here is the output from a run of the same program under Python 3.1.3 on MacOS 10.6:

The threads finish in apparently random order over six seconds

```
AirHead:src sholden$ python3 thread.py
Threads started
Thread-3 finished
tick
Thread-6 finished
tick
Thread-1 finished
Thread-4 finished
Thread-5 finished
tick
Thread-2 finished
tick
All threads done
AirHead:src sholden$
```

The simple expedient of removing the lock acquisition allows the threads to sleep in parallel, and without the limitation that only one thread can sleep at a time, all threads have terminated before the first (and last) tick from the main thread. Because the sleeps are intermingled, and again subject to random timing variations, the order of the threads finishing is unpredictable. [You should verify this assertion by making several runs of your program].

Removing the locks means one thread need not wait for others

```
"""
thread.py: Without threading.Lock, threads sleep in parallel.
"""
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, lock, *args, **kw):
        threading.Thread.__init__(self, *args, **kw)
        self.lock = lock
    def run(self):
        for i in range(10):
            self.lock.acquire()
            time.sleep(0.1)
            self.lock.release()
        self.lock.acquire()
        print(self.name, "finished")
        self.lock.release()

lock = threading.Lock()
bgthreads = threading.active_count()
tt = [MyThread(lock) for i in range(6)]
for t in tt:
    t.start()
print("Threads started")
while threading.active_count() > bgthreads:
    time.sleep(2)
    print("tick")
print("All threads done")
```

Now the six threads are all sleeping pretty much in parallel, and so all terminate after one second. The main thread therefore ticks once and sees all threads already terminated, and so the program ends after two seconds. Again you should find that the order in which the "worker" threads terminate is unpredictable, because of uncontrollable timing differences. Now it is much more likely that different threads could be printing at the same time, which could lead to garbled output, so we use the locks to ensure this cannot happen. A typical output follows.

It's all over before the first tick!

```
Threads started
Thread-4 finished
Thread-5 finished
Thread-6 finished
Thread-2 finished
Thread-1 finished
Thread-3 finished
tick
All threads terminated
```

Note

Interactive threading experiments can be tricky in IDEs: you may find, if you experiment with threads from the Ellipse interactive console, that output from a thread running in the background does not always appear immediately. This is because the IDE controls output in an attempt to ensure that your input is never interspersed with output from running code (which would make sessions extremely difficult to understand). So frequently you need to press **Enter** at the ">>> " prompt to allow output to become visible. A true interactive console session in a terminal window will not generally cause the same issues.

If you are starting to enjoy the possibilities opened up by the **threading** library, you should definitely look at its documentation to learn about **RIock**, **Condition**, **Semaphore** and **Event** objects.

The Queue Standard Library

This library was produced to provide programmers of threaded programs with a safe way for their threads to exchange

information. The **queue** module defines three classes that each have the same interface but queue things in slightly different ways. **queue.Queue** is a FIFO (first-in first-out) queue in which the first objects added to the queue are the first to be retrieved. This is the most usual type to use for handing out work to worker threads. **queue.LifoQueue** objects implement a stack of sorts. The next item retrieved is the most recently-added item. Finally, **queue.PriorityQueue** items are always retrieved in natural sort order.

When creating a queue, you can establish a maximum length for it by providing that length as an argument. If this maximum length is not provided, the queue will be of potentially infinite length, and further items may always be added to it. With a maximum length, there are only a given number of free slots, and attempts to add to a full queue will either block the thread that is attempting the add or raise an exception to show that the queue is full (or a combination of both). The thread-safety guarantees made by the library mean that the same queue item can be accessed by multiple threads without any need to lock the queue (locking as necessary is taken care of internally by the queue methods). When a queue is empty, any attempt to extract an item will either block or raise an exception (or both).

We are making only the simplest use of queues here, by using the **put()** and **get()** methods, to present a way of writing scalable threaded programs. There are many refinements you can adopt by reading the module documentation once you understand the basics. In threaded applications, simplest is almost always best, as most of us have brains that can only conceptualize a limited amount of parallelism and have difficulty predicting situations that cause problems in practice (such as deadlocks, where Thread A is blocked waiting for Thread B, which is blocked waiting for Thread A: since neither can progress, the two threads are doomed to wait for each other forever).

Adding Items to Queues: Queue.put()

queue.Queue.put(item, block=True, timeout=None) adds the given item to the queue. If **block** evaluates false, either the item is added immediately or an exception is raised. When **block** is True (the default case), either the item is added immediately or the putting thread blocks. *If timeout remains None, this could leave the thread blocked indefinitely in a non-interruptible state*. If a timeout (in seconds) is given, an exception will be raised if the item has not been added before the timeout expires.

Removing Items from Queues: Queue.get()

queue.Queue.get(block=True, timeout=None) attempts to remove an item from the queue. If an item is immediately available, it is always returned. Otherwise, if **block** evaluates false, an exception is raised. When **block** evaluates true, the process blocks either indefinitely (when timeout is None) or until the timeout (in seconds) has expired, in which case an exception is raised if no item has arrived.

Monitoring Completion: Queue.task_done() and Queue.join()

Every time an item is successfully added to a queue with put(), a *task count* is incremented. *Removing an item with get() does not decrement the counter.* To decrement the counter, the removing thread should wait until processing is complete and then call the queue's task_done() method.

If a queue is expected to end up empty, a thread can declare itself interested in the queue's exhaustion by calling its join() method. *This method blocks the calling thread until all tasks have been recorded as complete*. You should be confident that threads are all going to terminate correctly before using this technique, since it can lead to indefinite delays.

A Simple Scalable Multi-Threaded Workhorse

We'll finish the lesson by building a fairly general framework to allow you to run programs with "any number" of threads (sometimes the system places limits on the number of threads you can create).

The idea is to have a control thread that generates "work packets" for a given number of worker threads (with which it communicates by means of a queue). The worker threads compute the necessary results, and deliver them to a final output thread (by means of a second queue) which displays the results. The structure is quite general: work units can be generated by reading database tables, accepting data from web services, and the like. Computations can involve not only calculation but further database work or network communication, all of which can involve some (in computer terms) fairly extensive waiting.

The control thread is the main thread with which every program starts out (the *only* thread of all programs before these lessons). It creates an input and an output queue, starts the worker threads and the output thread, and thereafter distributes work packets to the worker threads until there is no more work. Since the worker threads are programmed to terminate when they receive None from the work queue, the control thread's final act is to Queue None for each worker thread and then wait for the queue to finally empty before terminating. The worker threads put a None to the output queue before terminating. The output thread counts these Nones, and terminates when enough None values have been seen to account for all workers.

The Output Thread

The output thread simply has to extract output packets from a queue where they are placed by the worker threads. As each worker thread terminates, it posts a None to the queue. When a None has been received from each thread, the output thread terminates. The output thread is told on initialization how many worker threads there are, and each time it receives another None it decrements the worker count until eventually there are no workers left. At that point, the output thread terminates. Create a new PyDev project named **Python4_Lesson11** and assign it to the **Python4_Lessons** working set. Then, in your **Python4_Lesson11/src** folder, create **output.py** as shown:

output.py: the output thread definition

```
"""
output.py: The output thread for the miniature framework.
"""
identity = lambda x: x

import threading
class OutThread(threading.Thread):
    def __init__(self, N, q, sorting=True, *args, **kw):
        """Initialize thread and save queue reference."""
        threading.Thread.__init__(self, *args, **kw)
        self.queue = q
        self.workers = N
        self.sorting = sorting
        self.output = []
    def run(self):
        """Extract items from the output queue and print until all done."""
        while self.workers:
            p = self.queue.get()
            if p is None:
                self.workers -= 1
            else:
                # This is a real output packet
                self.output.append(p)
        print("".join(c for (i, c) in (sorted if self.sorting else identity)(sel
f.output)))
        print ("Output thread terminating")
```

In this particular case, the output thread is receiving (index, character) pairs (because the workers pass through the position argument they are given as well as the transformed character, to allow the string to be reassembled no matter in what order the threads finish). Rather than output each one as it arrives, the output thread stores them until the workers are all done, then sorts them (unless sorting is disabled with **sorting=False**) and the characters extracted and joined together.

The Worker Threads

The Worker threads have been cast so as to make interactions easy. The work units received from the input queue are (index, character) pairs, and the output units are also pairs. The processing is split out into a separate method to make subclassing easier—simply override the process() method. Create **worker.py** as shown:

worker.py: the simple worker thread

```
"""
worker.py: a sample worker thread that receives input
           through one Queue and routes output through another.
"""
from threading import Thread

class WorkerThread(Thread):
    def __init__(self, iq, oq, *args, **kw):
        """Initialize thread and save Queue references."""
        Thread.__init__(self, *args, **kw)
        self.iq, self.oq = iq, oq
    def run(self):
        while True:
            work = self.iq.get()
            if work is None:
                self.oq.put(None)
                print("Worker", self.name, "done")
                self.iq.task_done()
                break
            i, c = work
            result = (i, self.process(c)) # this is the "work"
            self.oq.put(result)
            self.iq.task_done()
    def process(self, s):
        """This defines how the string is processed to produce a result"""
        return s.upper()
```

Although this particular worker thread is not doing particularly interesting processing (merely converting a single character to upper case), you can imagine more complex work units, perhaps with numerical inputs and the need for database lookup as well as interaction with local disk files.

The Control Thread

Everything is started off by the control thread (which imports the output and worker threads from their respective modules). It first creates the input and output queues. These are standard FIFOs, with a limit of 50% more than the number of worker threads to avoid locking up too much memory in buffered objects. Then it creates and starts the output thread, and finally creates and starts as many worker threads as configured by the WORKERS constant. Worker threads get from the input queue and put to the output queue. The control thread then simply keeps the input queue loaded as long as it can before sending the None values required to shut the worker threads down. Once the input queue is empty, the thread terminates.

control.py: The thread that drives everything else

```
"""
control.py: Creates queues, starts output and worker threads,
            and pushes inputs into the input queue.
"""
from queue import Queue
from output import OutThread
from worker import WorkerThread

WORKERS = 10

inq = Queue(maxsize=int(WORKERS*1.5))
outq = Queue(maxsize=int(WORKERS*1.5))

ot = OutThread(WORKERS, outq)
ot.start()

for i in range(WORKERS):
    w = WorkerThread(inq, outq)
    w.start()
instring = input("Words of wisdom: ")
for work in enumerate(instring):
    inq.put(work)
for i in range(WORKERS):
    inq.put(None)
inq.join()
print("Control thread terminating")
```

Running the program causes a prompt for input, which is then split up into individual characters and passed through the input queue to the worker threads. At present, ten threads operate in parallel, but the number can easily be varied by changing the definition of WORKERS in the source file. The output from a typical run is shown below.

A Bizarrely Complex Way to Covert a String to Upper Case?

```
Words of wisdom: Elemental forces are at work to change the way we live.
Worker Thread-2 done
Worker Thread-3 done
Worker Thread-4 done
Worker Thread-10 done
Worker Thread-9 done
Worker Thread-8 done
Worker Thread-11 done
Worker Thread-7 done
Worker Thread-5 done
Worker Thread-6 done
Control thread terminating
ELEMENTAL FORCES ARE AT WORK TO CHANGE THE WAY WE LIVE.
Output thread terminating
Control thread terminating
```

You will appreciate the need for the sorting if you study this output, from a typical run where the output thread was created with sorting=False:

Why sorting is required

```
Words of wisdom: Does the string really appear correct?
Worker Thread-7 done
Worker Thread-6 done
Worker Thread-4 done
Worker Thread-2 done
Worker Thread-11 done
Worker Thread-3 done
Worker Thread-9 done
Worker Thread-5 done
Worker Thread-10 done
Worker Thread-8 done
DOES THE STRING PEAELRYA PLAR CORERCT?
Output thread terminating
Control thread terminating
```

This ends our discussion of the queue.Queue object, and with it our somewhat lengthy study of threading.

Other Approaches

In the last two lessons, we've made use of the **threading** library module to write classes whose instances run as separate threads. If enough of these are started, the waiting that each thread has to do can be filled by useful work for other threads, and so a fairly high-bandwidth network channel can be kept busy and individual hold-ups can be made to matter much less. There are a number of other schemes that have been developed to control multiple asynchronous tasks.

The oldest (and the only one currently included in the standard library) is the asyncore module. With asyncore, each client process is a "channel," and you program the channels to respond to specific network events in specific ways. Asynchat is layered on top of asyncore and allows you to specify protocol handling by looking for specific sequences in the incoming data and triggering events when those sequences are detected.

The Twisted library is a system devised by Glyph Lefkowitz that has been used to good effect by many surprisingly large enterprises (including one business that has since been purchased by Google). Operations that will potentially block (cause the process to wait) return a **Deferred** object, which is effectively a promise of future data. A Deferred object is asked for its result by calling specific methods; if the data is not currently available, the Twisted scheduler suspends that activity until the Deferred request can be satisfied, and returns to some other suspended task that can now be restarted.

Stackless Python was an early attempt by Christian Tismer to allow massively parallel computing in Python by the provision of so-called "micro-threads." It has been used to great effect by a gaming company to provide a space "shoot-'em-up" environment for over 50,000 simultaneous players. More recent versions allow advanced capabilities like saving a computation on one computer and restoring it on another. This was very helpful in running code on a 250-CPU cluster.

A more recent approach to asynchronous networking is the Kamaelia package, initially developed by Michael Sparks for BBCResearch in the UK. Kamaelia, as far as I am aware, pioneered the use of generator functions to interact with the task scheduling environment. This approach has also been taken in Monocle, another even more recent development by Raymond Hettinger.

All in all, if you decide to venture beyond the standard library, a wealth of choices awaits you and not all of them rely on threading.

Multi-threading is one way to achieve asynchronous processing. For the CPython implementation (and others relying on single-processor guarantees to speed processing) this will not help if the application is CPU-bound, as all processing must take place on a single processor, and so the application cannot benefit from multiple processors in the computer it runs on.

Next, we'll go on to consider how to share work between multiple processes, which *can* be done on different processors and therefore extract more work from modern multi-processor hardware.

When you finish the lesson, don't forget to complete the homework!

Multi-Processing

Lesson Objectives

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

- use the Multiprocessing Library Module.
- create a Multiprocessing Worker Process Pool.

The Multiprocessing Library Module

The **multiprocessing** module was written specifically to offer features closely parallel to the **threading** library but allowing the individual threads of control to be processes rather than threads within a single process. This allows the operating system to take advantage of any parallelism inherent in the hardware design, since generally processes can run completely independently of one another, and on separate processors if they are available.

multiprocessing Objects

The **multiprocessing** library defines various classes, most of which operate in the same way as similar classes in the **threading** and related modules. Whereas in using **threading** you also imported resources from other modules, the **multiprocessing** module tries to put all necessary resources into one convenient place, simplifying imports. But you will easily recognize the program style from your recent work on multi-threading.

A Simple Multiprocessing Example

Our first multiprocessing example is marked up below as though we were editing the first *thread.py* example. This shows how similar the two environments are. Create a new pydev project named **Python4_Lesson12** and assign it to the **Python4_Lessons** working set. Then, in your **Python4_Lesson12/src** folder, create **process.py** as shown:

CODE TO TYPE: process.py

```
"""
process.py: demonstrate creation and parallel execution of processes.
"""

import multiprocessing
import time
import sys

def run(i, name):
    """Sleep for a given number of seconds, report and terminate."""
    time.sleep(i)
    print(name, "finished after", i, "seconds")
    sys.stdout.flush()

if __name__ == "__main__":
    for i in range(6):
        t = multiprocessing.Process(target=run, args=(i, "P"+str(i)))
        t.start()
    print("Processes started")
```

Note that this program has been correctly written as a module, so that the action of starting six processes is only performed by the process that runs this code, and not in any processes that may try to import the module. This is very important, because the subprocesses have to get their description of the work to be done from somewhere, and they do that by importing the main module. So in this case the subprocesses will import the **process** module (so the test **__name__ == "__main__"** is false) to access the **run()** function.

Note Not all platforms require that the main module be "importable" in that way. Since it does not hurt to write your programs this way, however, we recommend that you do so every time. Then, platform differences are less likely to "bite" you.

- The output should not be at all surprising:

Waiting in processes rather than threads

```
Processes started
P0 finished after 0 seconds
P1 finished after 1 seconds
P2 finished after 2 seconds
P3 finished after 3 seconds
P4 finished after 4 seconds
P5 finished after 5 seconds
```

A Multiprocessing Worker Process Pool

The lesson on multi-threading concluded with an example that used a pool of worker threads to convert the characters of a string into upper case. To demonstrate the (at least superficial) similarities between **multiprocessing** and **threading** and friends, we'll now adapt that code.

So first, copy the three programs (**output.py**, **worker.py**, and **control.py** from your **Python4_Lesson11/src** folder to your **Python4_Lesson12/src** folder.

The Output Process

The following listing shows the code for the **multiprocessor** version alongside the equivalent **threading**-based code. The differences are small enough to be negligible, and to allow anyone who understood the threaded code to also understand the multi-process version.

Modifying output.py for multi-processor operations

```
"""
output.py: The output process for the miniature framework.
"""
identity = lambda x: x

import multiprocessing
import sys

class OutThread(multiprocessing.Process):
    def __init__(self, N, q, sorting=True, *args, **kw):
        """Initialize process and save queue reference."""
        multiprocessing.Process.__init__(self, *args, **kw)
        self.queue = q
        self.workers = N
        self.sorting = sorting
        self.output = []

    def run(self):
        """Extract items and print until all done."""
        while self.workers:
            p = self.queue.get()
            if p is None:
                self.workers -= 1
            else:
                # This is a real output packet
                self.output.append(p)
        print("".join(c for (i, c) in (sorted if self.sorting else identity)(sel
f.output)))
        print ("Output process terminating")
        sys.stdout.flush()
```

The main difference between the two pieces of code is the use of **multiprocessing.process** in place of **threading.Thread**, and associated changes to a couple of comments. It is also necessary to flush the process's standard output stream to make sure that it is captured before the process terminates—otherwise you will see a confusing lack of output! (Feel free to try running the program with the flush() call commented out to verify this).

The Worker Process

The next listing shows the differences in the worker code when processes are being used instead of threads.

Modifying worker.py for multi-processor operations

```
"""
worker.py: a sample worker process that receives input
       through one queue and routes output through another.
"""

from multiprocessing import Process
import sys

class WorkerThread(Process):
    def __init__(self, iq, oq, *args, **kw):
        """Initialize process and save Queue references."""
        Process.__init__(self, *args, **kw)
        self.iq, self.oq = iq, oq
    def run(self):
        while True:
            work = self.iq.get()
            if work is None:
                self.oq.put(None)
                print("Worker", self.name, "done")
                self.iq.task_done()
                break
            i, c = work
            result = (i, self.process(c)) # this is the "work"
            self.oq.put(result)
            self.iq.task_done()
        sys.stdout.flush()
    def process(self, s):
        """This defines how the string is processed to produce a result."""

        return s.upper()
```

Again the only change is to use **Process** from **multiprocessing** instead of **Thread** from **threading**. (Two of the differences are again in comments.)

The Control Process

The control process again needs very little change: **queue** objects come from the **multiprocessing** module rather than the **queue** module, and in that module if you are going to **join()** a **queue** then you must use a **JoinableQueue**. The rest of the logic is exactly the same, with the exception that *the code must now be guarded so that it isn't executed when the module is imported by the multiprocessing module*. This means you have to indent the majority of the logic. This is easy in Ellipse: just highlight all the lines of code (making sure you are selecting whole lines) and then press **Tab** once.

Modifying control.py for multi-processor operations

```
"""
control.py: Creates queues, starts output and worker processes,
        and pushes inputs into the input queue.
"""
from multiprocessing import Queue, JoinableQueue
from output import OutThread
from worker import WorkerThread

if __name__ == '__main__':
    WORKERS = 10

    inq = JoinableQueue(maxsize=int(WORKERS*1.5))
    outq = Queue(maxsize=int(WORKERS*1.5))

    ot = OutThread(WORKERS, outq, sorting=True)
    ot.start()

    for i in range(WORKERS):
        w = WorkerThread(inq, outq)
        w.start()
    instring = input("Words of wisdom: ")
    # feed the process pool with work units
    for work in enumerate(instring):
        inq.put(work)
    # terminate the process pool
    for i in range(WORKERS):
        inq.put(None)
    inq.join()
    print("Control process terminating")
```

This version of control.py does exactly what the threading version did, except that the individual characters are now being passed to one of a pool of *processes* rather than one of a pool of threads. The computation is trivial, but the principle would be the same if the work packets were filenames and the outputs were MD5 checksums of the contents of the file (which could require substantial computation and I/O in the case of long files). Since the processes run independently of each other, they can be run on different processors at the same time, allowing programs to take true advantage of hardware parallelism. The output will seem prosaic for the amount of work that is being done!

Output of the multiprocessing upper-case converter

```
Words of wisdom: No words of wisdom at all, in fact. Just a rather long and boring line
 of text.
Worker Thread-2 done
Worker Thread-3 done
Worker Thread-4 done
Worker Thread-5 done
Worker Thread-6 done
Worker Thread-7 done
Worker Thread-8 done
Worker Thread-9 done
Worker Thread-10 done
Worker Thread-11 done
Control thread terminating
NO WORDS OF WISDOM AT ALL, IN FACT. JUST A RATHER LONG AND BORING LINE OF TEXT.
Output thread terminating.
```

Do not make the mistake of thinking that this brief treatment has taught you all you need to know about multiprocessing. There are many more things to learn about it including, for example, limitations on what can be transmitted from process to process through a **multiprocessing.Queue**. These restrictions are fairly commonsense, and are the result of having to pickle the objects to transmit them to the remote process. As long as you stick to Python's basic data objects (and combinations thereof), you should be fine. Other restrictions are less obvious: when you subclass **multiprocess.Process**, the instances should be pickleable (because the class has to be instantiated in a new process when the instance's start() method is called).

As systems evolve, multiprocessor solutions will become more and more common, and it will be necessary to put systems together to take control of multi-processor machines. This lesson is intended to give you the necessary grounding so that you can take the next steps with confidence.

When you finish the lesson, don't forget to complete the homework!

Functions and Other Objects

Lesson Objectives

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

- interact with more Functions.
- employ more Magic Methods.

A Deeper Look at Functions

Required Keyword Arguments

You already know that the arguments passed to a function call must match the parameter specifications in the function's definition. Any mismatch can be taken up in the definition, where a parameter of the form *name associates unmatched positional arguments with a tuple and one of the form **name associates the names and values of unmatched keyword arguments with the keys and values of a dict.

You have also seen that a positional argument may be associated with a keyword parameter and vice versa. You currently have no way, however, of requiring that specific arguments be presented as keyword arguments. You can specify such a requirement by inserting an asterisk on its own as a parameter specification: any parameters that follow the star (other than the *args and **kwargs arguments, if present) *must* be provided as keyword arguments on the call.

Investigating this phenomenon is quite easy in the interactive console:

Investigating function signatures

```
>>> def f(a, *, b, c=2):
...     print("A", a, "B", b, "C", c)
...
>>> f(1, 2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
>>> f(1, c=3)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: f() missing 1 required keyword-only argument: 'b'
>>> f(1, b=2, c=3)
A 1 B 2 C 3
>>> f(1, b=2)
A 1 B 2 C 2
>>>
```

Attempting to provide a positional argument for b raises an exception because of the wrong number of positional arguments. The second test is the most telling one, as that explains the *requirement* for a keyword argument b.

Function Annotations

We mention this feature because you may come across some code that uses it, and wonder what on Earth is going on. In Python 3, functions and their parameters can be *annotated*. A parameter is annotated by following its name with a colon and an expression, and a function is annotated by following its parameter list with "->" and an expression.

The language definition specifically avoids associating any kind of meaning to annotations. The stated intention is that if people find ways of using annotations that find general acceptance, specific semantics may be added to the interpreter at a later date; for now you can access them through the **__annotations__** attribute of the function object. This is a dict in which each of the function's annotated parameters is stored against the parameter name as key. The function's return-value annotation, if present, is stored against key "return" which, being a Python keyword, cannot be the name of any parameter.

Just to show you how annotations appear in practice, we'll create an annotated function in an interactive interpreter session:

INTERACTIVE SESSION:

```
>>> def f(i: int, x:float=1.2) -> str:
...     return str(i*x)
...
>>> f.__annotations__
{'i': <class 'int'>, 'x': <class 'float'>, 'return': <class 'str'>}
>>>
```

Although there is no restriction on the expressions used as annotations, in practice most people see them as being useful for making assertions about the types of arguments and the function's return value. At present, nothing in the interpreter uses the anotation information at all. You would need to specifically action such uses with additional code if you don't want your annotation data to be ignored. It is likely that, as the feature becomes better known, frameworks will emerge to make use of different types of annotation data.

Nested Functions and Namespaces

Although you have seen functions with function definitions inside them, we have not yet formalized the rules for looking up names within those functions. You already know the general rule for (unqualified) name resolution in Python: first look in the local namespace, then look in the (module) global namespace, and finally look in the built-in namespace.

The only additional complexity that nested functions introduce is that the local namespace is actually enhanced by names from surrounding functions (unless they are redefined in the contained function). Remember that *a name is only considered local to a function if the name is bound in that function*. So when a function is defined inside a function, a name can be a reference from the function call's namespace, or a reference to the namespace of the function call during which the inner function was defined, and this regress can go on until the outermost function call is encountered.

Understanding Python as you do now, you will see that it requires some trickery to allow a function to return another function defined inside the first function. That is because the returned function may contain references to values defined in the local namespace of the (now completed) function call that returned it! We do not need to examine the mechanism the interpreter uses to resolve this issue, but since it is a genuine feature of the language, it is one that every implementation has to solve in its own way.

Python 3 also introduces a second declaration statement, the **nonlocal** statement. This can be used to force an apparently local variable to instead be treated as though it came from the containing scope where it is already defined. This is slightly different from the *global* statement, in that the interpreter searches the containing scopes (function namespaces) to locate the one that already contains a definition of the name(s) listed after the **nonlocal** keyword. (The **global** statement always and unambiguously places the name in the module global namespace, whether it has been defined there or not).

Create a new PyDev project named **Python4_Lesson13** and assign it to the **Python4_Lessons** working set. Then, in your **Python4_Lesson13/src** folder, create **nonloc.py** as shown:

Difference between global and nonlocal: create this as nonloc.py

```
a, b, c = "Module a", "Module b", "Module c"
def outer():
    def inner():
        nonlocal b
        global c
        a = "Inner a"
        b = "Inner b"
        c = "Inner c"
        print("inner", a, b, c)
    a = "Outer a"
    b = "Outer b"
    c = "Outer c"
    print("outer", a, b, c)
    inner()
    print("outer", a, b, c)

print("module", a, b, c)
outer()
print("module", a, b, c)
```

- Save and run it:

The result of running nonloc.py

```
module Module a Module b Module c
outer Outer a Outer b Outer c
inner Inner a Inner b Inner c
outer Outer a Inner b Outer c
module Module a Module b Inner c
```

Just as the **global** statement allows the inner() function to refer to the module-global "c" name, so the **nonlocal** statement allows it to use the name "b" to refer to the outer function's "b." After the call to outer(), only the module-global "c" has changed, because only "c" was declared as **global** in the inner() function.

Partial Functions

You learned about the **functools** module when we were discussing decorators earlier in this course. The module contains another useful function that allows you to take a function and define another function that is the same as the first function, *but with fixed values for some arguments*. The signature of the function is:

functools.partial(f[, *args[, **kw]]) returns a function **g** which is the same as **f** with the positional arguments *args* giving values for the initial positional arguments and the keyword arguments *kw* setting default values for the given named arguments. The intention is to allow you to fix some arguments of a function, leaving you with a function-like object to which the remaining arguments can be applied at your convenience. The resulting partial function objects cannot be called with quite the same abandon as real functions, however, since certain counterintuitive behaviors can occur.

Partial function examples

```
>>> import functools
>>> def fp(a, b, c="summat", d="nowt"):
...     print("a b c d", a, b, c, d)
...
>>> fp("ayeup", "geddaht")
a b c d ayeup geddaht summat nowt
>>> fp1 = functools.partial(fp, 1, b=2)
>>> fp1()
a b c d 1 2 summat nowt
>>> fp1("ayeup", "geddaht")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: fp() got multiple values for argument 'b'
>>> fp1(c="ayeup", d="geddaht")
a b c d 1 2 ayeup geddaht
>>> fp2 = functools.partial(fp, 1, c="two")
>>> fp2("ayeup", "geddaht")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: fp() got multiple values for argument 'c'
>>> fp2
functools.partial(<function fp at 0x000000000349B2F0>, 1, c='two')
>>> fp2("ayeup", c="geddaht")
a b c d 1 ayeup geddaht nowt
>>>
```

fp1 is ostensibly a function taking two keyword arguments (its two positionals having been applied in the creation of the partial). The expression **fp1("ayeup", "geddaht")**, however, makes it plain that the first positional argument is being provided to match up with **fp()**'s *b* argument, and that when the same keyword argument is later applied a duplication is detected.

The simplest solution to this dilemma is to always replace positional parameters with positional arguments and replace keyword parameters with keyword arguments when using partial(). This rule also has to be extended to the calls of the partial functions. The first call to **fp2()** shows that although the partial function has one positional and one keyword parameter, it is not possible to match a positional argument to the keyword parameter *d* as would be possible with a real function. So remember to treat partials carefully when you encounter them.

One very nice little example from the documentation shows how a default can be applied to a required argument. The int() built-in type can be called with a number or a string as an argument. When called with a string, a second argument *base* can be provided which determines the number system used to interpret the string. Providing that argument creates a partial object that will convert base-2 strings to integers.

Partial(int) function converts binary strings

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = "Convert base-2 string to int."
>>> basetwo("1111")
15
>>> basetwo("1001010")
74
>>> help(basetwo)
Help on partial object:

class partial(builtins.object)
 |  partial(func, *args, **keywords) - new function with partial application
 |  of the given arguments and keywords.
 |
 |  Methods defined here:
 |
 |  __call__(self, /, *args, **kwargs)
 |      Call self as a function.
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __reduce__(...)
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |
 |  args
 |      tuple of arguments to future partial calls
 |
 |  func
 |      function object to use in future partial calls
 |
 |  keywords
 |      dictionary of keyword arguments to future partial calls
```

Beware of the differences between partial objects and true functions, and respect them. While partials can be very helpful, they are only a shorthand and not a complete replacement.

More Magic Methods

We have explained in the past how certain operations and functions cause the interpreter to invoke various "magic" methods—methods whose names usually start and end with a double underscore, causing some people to refer them as "dunder methods." In particular you should now be aware of the attribute access methods (__getattr__(), __setattr__(), and __delattr__()) and the indexing methods (__getitem__(), __setitem__(), and __delitem__(), which parallel the attribute access methods but operate on mappings rather than namespaces (and can also be used to index lists and other sequences, with slice objects as arguments where necessary).

Now we are going to cover a few more of those magic methods and explain a little more about the interpreter's interfaces to the various objects you can create. Understanding in this area allows you to take advantage of the natural operation of the interpreter. It's a little like jiu-jitsu: you write your objects to fit in with the way the interpreter naturally does things rather than trying to overpower the interpreter.

How Python Expressions Work

This simplified treatment expresses the way that the interpreter works to a first approximation. As always, we try to be as precise as possible without necessarily providing *exact* detail of what goes on in the more complex corner cases.

When you see the expression **s = x + y** in a program, the interpreter has to decide how to evaluate it. It does so by looking for specific methods on the **x** and **y** objects. For addition, the relevant methods are **__add__()** and **__radd__()**. First the interpreter looks for an **x.__add__()** method (special/magic methods are always looked up on the class and its parents, never on the instance). If such a method exists. **x.__add__(y)** is called. If this call returns a result, that becomes the value of the expression.

The method may, however, choose to indicate that it is unable to compute a response (for example because **y** is incompatible) by returning a special built-in value **NotImplemented**. In that case, the interpreter next looks for a **y.__radd__()** method ("radd" is intended to be a mnemonic for "reflected add"). If such a method exists, **y.__radd__(x)** is called and, unless it returns **NotImplemented**, the return value becomes the value of the expression. There is one exception to this rule: if the two values are of the same type, the **__radd__()** method is not called. The assumption is that if **a** and **b** are of the same type and you can't (say) add **a** to **b**, then you shouldn't be able to add **b** to **a** either, and there is no point trying.

Try it out in an interactive session:

Verifying use of reflected operators

```
>>> class mine:
...     def __add__(self, other):
...         print("__add__({}, {})".format(self, other))
...         return NotImplemented
...     def __radd__(self, other):
...         print("__radd__({}, {})".format(self, other))
...         return 42
...     def __repr__(self):
...         return "[Mine {}]".format(id(self))
...
>>> class yours:
...     def __add__(self, other):
...         print("__add__({}, {})".format(self, other))
...         return NotImplemented
...     def __radd__(self, other):
...         print("__radd__({}, {})".format(self, other))
...         return NotImplemented
...     def __repr__(self):
...         return "[Yours {}]".format(id(self))
...
>>> m1 = mine()
>>> m2 = mine()
>>> m1, m2
([Mine 4300644112], [Mine 4300643600])
>>> y1 = yours()
>>> y2 = yours()
>>> y1, y2
([Yours 4300644240], [Yours 4300643728])
>>>
>>> m1+m2
__add__([Mine 4300644112], [Mine 4300643600])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'mine' and 'mine'
>>> y1+y2
__add__([Yours 4300644240], [Yours 4300643728])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'yours' and 'yours'
>>> m1+y2
__add__([Mine 4300644112], [Yours 4300643728])
__radd__([Yours 4300643728], [Mine 4300644112])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'mine' and 'yours'
>>> y1+m2
__add__([Yours 4300644240], [Mine 4300643600])
__radd__([Mine 4300643600], [Yours 4300644240])
42
>>>
```

As you can see, since both classes' **__add__()** methods return **NotImplemented**, attempting to add a **mine** to a **mine** or a **your** to a **your** will fail, raising an exception. The third case also raises an exception because the **__radd__()** method of the **yours** right-hand operand also returns the value **NotImplemented**. The final test works, however, because **mine.__radd__()** actually returns a value (albeit one that does not depend on its operands at all).

There is another series of special methods associated with the augmented arithmetic operations (that is, "+=", "-=" and so on). When you see a statement such as **x += y** (that is to say, any statement using augmented assignment operations) in a program, the interpreter evaluates it by looking for a specific method on the **x** object. For addition, the relevant method is **__iadd__()**. If this method does not exist, the statement is treated as though it read **x = x+y**. If the **x.__iadd__()** method is found, however, it is called with **y** as an argument, and the result (which may be a modified version of the existing object or a completely new object, entirely at the option of the implementor of the object in question) is bound to **x**. Following are the methods

corresponding to the basic Python arithmetic operations.

Operator	Standard Method	Reflected Method	Augmented Method
+	__add__()	__radd__()	__iadd__()
-	__sub__()	__rsub__()	__isub__()
*	__mul__()	__rmul__()	__imul__()
/	__truediv__()	__rtruediv__()	__itruediv__()
//	__floordiv__()	__rfloordiv__()	__ifloordiv__()
%	__mod__()	__rmod__()	__imod__()
divmod()	__divmod__()	__rdivmod__()	__idivmod__()
**	__pow__()	__rpow__()	__ipow__()
<<	__lshift__()	__rlshift__()	__ilshift__()
>>	__rshift__()	__rrshift__()	__irshift__()
&	__and__()	__rand__()	__iand__()
^	__xor__()	__rxor__()	__ixor__()
|	__or__()	__ror__()	__ior__()

So you now understand a little more about functions in Python, and understand more of the role of "magic" methods in Python.

In the next lesson, we consider some of the differences between small projects and large ones.

When you finish the lesson, don't forget to complete the homework!

Context Managers

Lesson Objectives

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

- use another Python Control Structure called the With Statement.
- use Decimal Arithmetic and Arithmetic Contexts in Python.

Another Python Control Structure: The With Statement

One of the more recently added control constructs in Python is the **with** statement. This allows you to create resources for the duration of an indented suite and have them automatically released when no longer required. The statement's basic syntax is:

with statement syntax

```
with object1 [as name1][, object2 [as name2]] ...:
   [indented suite]
```

The **objects** are referred to as *context managers*, and if the indented suite needs to refer to them, they can be named in the **as** clause(s) (which can otherwise be omitted). Nowadays, files are context managers in Python, meaning that it is possible to write file processing code without explicitly closing the files you open.

Using a Simple Context Manager

Create the usual project folder (**Python4_Lesson14**) and assign it to the **Python4_Lessons** working set. In your **Python4_Lesson14** folder, create a file named **localtextfile**. Then, open an interactive console session and enter commands as shown:

The following interactive console session shows how to use files as context managers.

An Introduction to Context Managers

```
>>> with open(r"v:\workspace\Python4_Lesson14\src\localtextfile") as f:
...     print("f:", f)
...     print("closed:", f.closed)
...     for line in f:
...         print(line, end='')
...
f: <_io.TextIOWrapper name='v:\\workspace\\Python4_Lesson14\\src\\localtextfile'
 mode='r' encoding='cp1252'>
closed: False
>>> f
<_io.TextIOWrapper name='v:\\workspace\\Python4_Lesson14\\src\\localtextfile' mo
de='r' encoding='cp1252'>
>>> f.closed
True

>>> f = open(r"v:\workspace\Python4_Lesson14\src\localtextfile", 'r')
>>> 3/0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError: division by zero
>>> f
<_io.TextIOWrapper name='v:/workspace/Python4_Lesson14/src/localtextfile' mode='
r' encoding='cp1252'>
>>> f.closed
False
>>> with open(r"v:\workspace\Python4_Lesson14\src\localtextfile") as f:
...     3/0
...
Traceback (most recent call last):
  File "<console>", line 2, in <module>
ZeroDivisionError: division by zero
>>> f.closed
True
>>>
```

You can see that the **with** statement is a way of controlling the context of execution for the controlled suite. You might wonder why we didn't simply bind the Python file object (the result of opening the file) using an assignment statement. The major purpose of using **with** in this case is to ensure that, if anything goes wrong inside the context-controlled indented suite, the file will be correctly closed (similarly to the way it might be in the **finally** clause of a **try ... finally** statement.

Files, in and out of context

```
>>> with open(r"v:\workspace\Python4_Lesson14\src\localtextfile") as f:
...     print("f:", f)
...     print("closed:", f.closed)
...     for line in f:
...         print(line, end='')
...
f: <_io.TextIOWrapper name='v:\\workspace\\Python4_Lesson14\\src\\localtextfile'
 mode='r' encoding='cp1252'>
closed: False
The open function returns a file object.
This has an __enter__() method that simply
returns self. Its __exit__() method calls
its __close__() method.
>>> f
<_io.TextIOWrapper name='v:\\workspace\\Python4_Lesson14\\src\\localtextfile' mo
de='r' encoding='cp1252'>
>>> f.closed
True
>>> f = open(r"v:\workspace\Python4_Lesson14\src\localtextfile", 'r')
>>> 3/0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError: division by zero
>>> f
<_io.TextIOWrapper name='v:\\workspace\\Python4_Lesson14\\src\\localtextfile' mo
de='r' encoding='cp1252'>
>>> f.closed
False
>>> f.close()
>>> with open(r"v:\workspace\Python4_Lesson14\src\localtextfile") as f:
...     3/0
...
Traceback (most recent call last):
  File "<console>", line 2, in <module>
ZeroDivisionError: division by zero
>>> f.closed
True
>>>
```

In the first **with** example, we saw that f was a **standard IO Wrapper object** (in point of fact, exactly the same object returned by the **open()** call, though as you will learn this is *not* typical of context managers). When the indented suite is run, the file is initially open. Next we see that the file object (still available after the **with**) is closed when the **with** statement terminates, *even though no explicit action was taken to close it*. You will understand this after the next interactive session.

Next you reminded yourself that **when an exception occurs during regular file processing** the file remains open unless explicit action is taken to close it. When the **exception occurs inside the suite of the with statement**, however, once again we see that the file is magically closed without any explicit action being taken. The magic is quite easily explained (as usual in Python, where a simple, easy-to-understand style is preferred) by two file magic methods we have not previously discussed.

The Context Manager Protocol: __enter__() and __exit__()

The **with** statement has rules for interacting with the object it is given as a context manager. It processes **with expr** by evaluating the expression and saving the resulting *context manager object*. The context manager's __enter__() method is then called, and if the **as name** clause is included, the result of the method call is bound to the given name. Without the **as name** clause, the result of the __enter__() method is not available. The indented suite is then executed.

As the execution of the suite progresses, an exception may be raised. If so, the execution of the suite ends and the context manager's __exit__() method is called with three arguments together referencing detailed information about the causes and location of the exception.

If no exception is raised and the suite terminates normally (that is, by "dropping off the end"), the context manager's __exit__() method is called with three **None** arguments.

There are other ways that the **with** suite can be exited, all fairly normal—how many ways can you think of? In those circumstances, the context manager's __exit__() method is called with three **None** arguments, and then the normal exit is taken.

The reason for the name "context manager" is that the indented suite in a with statement is surrounded by calls to the manager's __enter__() and __exit__() methods, which can therefore provide some context to the execution of the suite. Note carefully that the __exit__() method is *always* called—*even when the suite raises an exception*.

Writing Context Manager Classes

As is so often the case in Python, it is quite easy to write a class that demonstrates exactly how the context manager objects work with the interpreter as it executes the **with** statement. Since there are two alternative strategies for handling the raising of an exception in the indented suite, an __init__() method can record in an instance variable which strategy the creator (the code calling the class) chooses. If no exception is raised, this will make no difference.

Besides the very simple __init__() outlined (which is not itself a part of the context manager protocol), you just need the __enter__() and __exit__() methods. If you are only interested in finding out how the **with** statement works, these methods don't have to do a lot except print out useful information. Try this out in an interactive interpreter session:

Investigating the with Statement

```
>>> class ctx_mgr:
...     def __init__(self, raising=True):
...         print("Created new context manager object", id(self))
...         self.raising = raising
...     def __enter__(self):
...         print("__enter__ called")
...         cm = object()
...         print("__enter__ returning object id:", id(cm))
...         return cm
...     def __exit__(self, exc_type, exc_val, exc_tb):
...         print("__exit__ called")
...         if exc_type:
...             print("An exception occurred")
...             if self.raising:
...                 print("Re-raising exception")
...             return not self.raising
...
>>> with ctx_mgr(raising=True) as cm:
...     print("cm ID:", id(cm))
...
Created new context manager object 4300642640
__enter__ called
__enter__ returning object id: 4300469808
cm ID: 4300469808
__exit__ called
>>> with ctx_mgr(raising=False):
...     3/0
...
Created new context manager object 4300642768
__enter__ called
__enter__ returning object id: 4300469904
__exit__ called
An exception occurred
>>> with ctx_mgr(raising=True) as cm:
...     3/0
...
Created new context manager object 4300642640
__enter__ called
__enter__ returning object id: 4300469744
__exit__ called
An exception occurred
Re-raising exception
Traceback (most recent call last):
  File "<console>", line 2, in <module>
ZeroDivisionError: division by zero
>>>
```

Your context manager object does not get too much of a workout in the above session, but as always you should feel free to try out other things. You are unlikely to cause a fire or bring the server to a halt by being a little adventurous: you are now a seasoned Python programmer, and can (we hope) be trusted to flex your muscles a little. Let's just review the output from that session:

What Just Happened?

```
>>> with ctx_mgr(raising=True) as cm:
...     print("cm ID:", id(cm))
...
Created new context manager object 4300642640
__enter__ called
__enter__ returning object id: 4300469808
cm ID: 4300469808
__exit__ called
>>> with ctx_mgr(raising=False):
...     3/0
...
Created new context manager object 4300642768
__enter__ called
__enter__ returning object id: 4300469904
__exit__ called
An exception occurred
>>> with ctx_mgr(raising=True) as cm:
...     3/0
...
Created new context manager object 4300642640
__enter__ called
__enter__ returning object id: 4300469744
__exit__ called
An exception occurred
Re-raising exception
Traceback (most recent call last):
  File "<console>", line 2, in <module>
ZeroDivisionError: division by zero
>>>
```

In the **first example**, you can see that this context manager returns an entirely different object as the result of its __enter__() method. The print statement which forms the indented suite demonstrates that the name **cm** is bound in the **with** statement to the result of the context manager's __enter__() method and not the context manager itself. (The file open() example earlier is atypical, as a file object's __enter__() method returns **self**). No exception is raised by the indented suite, and so the __exit__() method simply reports it has been called.

The **second example** raises an exception in the context of a context manager that was created *not* to re-raise the exception. So it does report the fact that an exception was raised, but then it again terminates normally (because its **self.raising** attribute has the value **False**, and so the method returns **True**).

The **third example** is exactly the same as the second except that the instance is created with its **raising** attribute **True**. This means that once the instance has reported the exception it announces its intention to re-raise it, and does so by returning **False**.

Library Support for Context Managers

Although you have just seen it is very easy to write a simple context manager class, it can be even easier to use context managers if you use the **contextlib** module. This contains a decorator called **contextmanager** that you can use to create context managers really simply. There is no need to declare a class with __enter__() and __exit__() methods.

You must apply the **contextlib.contextmanager** decorator to a generator function that contains precisely one **yield** expression. When the decorated function is used in a **with** statement, the (decorated) generator's **next** method is called for the first time, so the function body runs right up to the **yield**. The yielded value is returned as the result of the context manager's __enter__() method, and the indented suite of the **with** statement then runs.

If the indented suite raises an exception, it appears inside the context manager as an exception raised by the **yield**. Your context manager can choose to handle the exception (by processing the **yield** as part of the indented suite of a **try** statement) or not (in which case the exception must be re-raised after logging or other actions if the surrounding logic is to see it). So your context manager can trap exceptions raised by the indented suite and suppress them simply by choosing not to re-raise them.

Experimenting with contextlib.contextmanager

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def ctx_man(raising=False):
...     try:
...         cm = object()
...         print("Context manager returns:", id(cm))
...         yield cm
...         print("With concluded normally")
...     except Exception as e:
...         print("Exception", e, "raised")
...         if raising:
...             print("Re-raising exception")
...             raise
...
>>> with ctx_man() as cm:
...     print("cm from __enter__():", id(cm))
...
Context manager returns: 4300470512
cm from __enter__(): 4300470512
With concluded normally
>>> with ctx_man(False) as cm:
...     3/0
...
Context manager returns: 4300801264
Exception division by zero raised
>>> with ctx_man(True) as cm:
...     3/0
...
Context manager returns: 4300801280
Exception division by zero raised
Re-raising exception
Traceback (most recent call last):
  File "<console>", line 2, in <module>
ZeroDivisionError: division by zero
>>>
```

This interactive session shows that it is possible to create equivalent context managers using this approach. The same parameterization of the functionality is provided (so you can say when creating the context manager whether or not it should re-raise exceptions). **contextlib.contextmanager** provides a nice compromise between writing a full context manager and using older, less well-controlled methods (such as **try ... except ... finally**) of controlling the execution context. You will find that the other members of the **contextlib** library can also be useful in creating and supporting context managers.

Nested Context Managers

The statement:

OBSERVE:

```
with expr1 as name1, expr2 as name2:
    [indented suite]
```

is equivalent to:

OBSERVE:

```
with expr1 as name1:
    with expr2 as name2:
        [indented suite]
```

This shows that the **expr1** context wraps the **name2** context. If an exception occurs in the indented suite, it will present as a call to expr2.__exit__() with the necessary exception-related arguments. As always, the

__exit__() method has the choice of returning **True** (which suppresses the exception, resulting in a call to **expr1.__exit__()** with three **None** arguments) or **False**, in which case the exception is automatically re-raised and **expr1.__exit__()** is called with the traceback arguments. It also has the choice of returning **True** to suppress the exception or **False** to re-raise it a second time.

The multi-context form of the **with** statement is a simple syntactic convenience; no new functionality is introduced, but it does reduce the indentation level required for the indented suite. This enhances readability without compromising simplicity.

Decimal Arithmetic and Arithmetic Contexts

Decimal arithmetic is quite a large topic, and we don't cover it anywhere near fully in this chapter. The **decimal** module was designed to allow easy decimal calculations, which are much more appropriate when accurate answers are required than the sometimes-slightly-inaccurate floating-point numbers built into the language. This is typically the case in commerce and accounting, where strict decimal arithmetic has been used for hundreds of years and inaccuracies in representation cannot be permitted.

Note

Fixed-point vs. floating-point. In fixed-point representations, a digit in a given position always has a specific value. Thus in the number represented as "3.14159", the digit after the decimal point always represents some number of tenths, and the given fixed-point representation can represent numbers between -9.9999 and +9.9999, with the smallest difference beteen two numbers being 0.0001 (which is the difference between every pair of "adjacent" numbers). Floating-point representations allow the point (in this case, the decimal point) to move. This means that the size of the numbers you can represent is independent of the number of digits of precision you can represent, and depends primarily on the range of exponents. If we allow exponents to range from -5 to +5, with five digits the smallest positive number you can represent is 0.00001 * 10 ^ -5 (which is 0.0000000001) and the largest is 0.99999 * 10 ^ 5 (or 99999.0). But the gaps between the adjacent *larger* numbers are much greater than the gaps between the *smaller* numbers. The value 0.99999 * 10 ^ 5 is conventionally written as 0.99999E5.

Decimal Arithmetic Contexts

This section will briefly introduce the **decimal** module, to whose documentation you are referred for further information. The context in which decimal arithmetic is performed has several elements:

Attribute	Meaning
prec	Specifies *precision*—how many digits are retained in calculations (the default is 28 decimal digits). The decimal point may occur many places before or after the significant digits, since decimal arithmetic can handle a floating decimal point. **decimal** knows how to maintain proper precision through calculations, so for example**Decimal("2.50") * Decimal("3.60")** evaluates to **Decimal("9.0000")**.
rounding	One of a set of constants defined in the **decimal** module that tells the arithmetic routines how to round when precision must be discarded.
flags	A list of *signals* (discussed below) whose flags are currently set. Flags are usually clear when a context is created, and set by abnormal conditions in arithmetic operations, although they can be set when the context is created if required.
traps	A list of signals whose setting by an arithmetic operation should cause an exception to be raised.
Emin	An integer containing the *minimum* value the exponent is allowed to take. This sets a lower bound on the values that numbers can represent.
Emax	An integer containing the *maximum* value the exponent is allowed to take. This sets an upper bound on the values that the numbers can represent.
capitals	**True** (the default) to use an upper-case "E" in exponential representations, **False** to use a lower case "e".
clamp	**True** (the default) to ensure that numbers are represented as ten to the power of the exponent times some number in the range 0.1 <= mantissa < 1.0. This ensures easy interchange with other computers using standard "IEEE 754" decimal representation. **False** allows some latitude in representation, allowing a wider range of numbers with fewer digits of actual precision at the cost of losing "IEEE normalization" at the extremes of the value range.

The **decimal** module has been carefully written to ensure that each thread can have an independent decimal context (because it would be disastrous if one thread could affect another by making changes to a shared

context).

Most of the attributes of the context are fairly esoteric stuff that you really don't need to alter. For many applications, you can just use the default context. While **prec** and **rounding** are fairly frequently adjusted, **capitals** and **clamp** are rarely touched.

Decimal Signals

Certain things can happen during arithmetic operations that cause the results to be imprecise or otherwise misleading, and the operations raise signals to indicate this. The **decimal** code responds to these signals by setting flags in the arithmetic context. If the trap corresponding to a signal is set, an exception is raised after the flag is set. The following flags are defined:

Signal	Raised when ...
Clamped	When a number's representation had to be modified to normalize it to a mantissa range of 0.1 to 0.999999999999...
DecimalException	Not raised: this is simply a base class for the others, and a subclass of the built-in ArithmeticError exception.
DivisionByZero	Either a division or a modulo operation had a left operand of zero.
Inexact	Indicates that rounding took place after an operation.
InvalidOperation	This often occurs when operations are performed on decimal infinities or "Not a Number" objects.
Overflow	The result cannot be represented with an exponent Emax or less.
Rounded	Rounding has occurred. If the digits rounded were all zero, no information has been lost.
Subnormal	The number cannot be represented with an exponent of Emin or larger.
Underflow	The result of an arithmetic operation was so small in magnitude that the most accurate way to represent it is as 0.

The Default Decimal Context

You can access the default decimal context using the **getcontext()** function from the **decimal** module. Contexts know how to present themselves in a fairly readable form, and you can modify the context just by assigning to its various attributes. You can also create copies of contexts and switch between them. Finally, of course, you can create instances of the **decimal.Context** class, providing the non-default required attributes as keyword arguments. Note that if you modify **decimal.DefaultContext**, it will change the default values used to create future contexts. This is useful for setting up defaults before creating multiple threads, but should not be used casually in non-threaded programs.

Understanding Decimal Contexts

```
>>> from decimal import *
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> getcontext()
Context(prec=60, rounding=ROUND_HALF_DOWN, Emin=-999999, Emax=999999, capitals=1
, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')
>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1,
 clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1,
 clamp=0, flags=[], traps=[])
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')
>>> setcontext(BasicContext)
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_UP, Emin=-999999, Emax=999999, capitals=1, c
lamp=0, flags=[], traps=[Clamped, InvalidOperation, DivisionByZero, Overflow, Un
derflow])
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
decimal.DivisionByZero: [<class 'decimal.DivisionByZero'>]
>>> with localcontext() as ctx:
...     ctx.prec = 42
...     s = Decimal(1) / Decimal(7)
...     print(s)
...
0.142857142857142857142857142857142857142857
>>> s = +s
>>> print(s)
0.142857143
>>>
```

You can see that the **decimal** module provides a number of "ready-made" contexts, which can be modified easily by attribute assignment. It is easy to make changes to the current context's attributes, but these changes are permanent. The **decimal.localcontext()** function returns a context manager that sets the active thread's current context to the context provided as an argument or (in the case above where no argument is provided) the current context. The **with** statement provides a natural way to perform such localised changes. Note that the unary plus sign in "**+s**" does actually perform a conversion, because it is an arithmetic operation whose result must be conditioned by the (now restored) original context.

With context managers and the **with** statement, Python gives you the chance to closely control the context of execution of your code. You should consider them whenever you might consider **try ... except ... finally**.

You are getting close to the end of the Certificate Series in Python! Well done! Keep it up!

When you finish the lesson, don't forget to complete the homework!

Memory-Mapped Files

Lesson Objectives

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

- utilize Memory Mapping.
- process a Memory-Mapped Example.

Memory Mapping

Files can be so large that it is impractical to load all of their content into memory at once. The **mmap.mmap()** function creates a virtual file object. Not only can you perform all the regular file operations on a memory-mapped file, you can also treat it as a vast object (far larger than any real object could be) that you can address just like any other sequence.

This technique deals with files by mapping them into your process's address space. The **mmap** module allows you to treat files as similar to **bytearray** objects—you can index them, slice them, search them with regular expressions and the like. Many of these operations can make it much easier to handle the data in a file: without memory mapping, you have to read the file in chunks and process the chunks (assuming the files are too large to read into memory as a single chunk). This makes it very difficult to process strings that overlap the inter-chunk boundaries. Memory mapping allows you to pretend that all the data is in memory at the same time even when that is not actually the case. The necessary manipulations to allow this are performed automatically.

In this lesson, we primarily cover only the details of **mmap** that apply across both Windows and Unix platforms, and a few Windows-specific features. You should be aware that there are different additional feature sets available for Windows and Unix platforms. The documentation on the module is fairly specific about the implementation differences.

Memory-Mapped Files Are Still Files

In standard file operations, there is no difference between a memory-mapped file and one that is opened in the regular way—all regular file access methods continue to work, and you can also treat the file content pretty much like a bytearray.

Here's a simple example from the module's documentation to get you started.

Getting Started with Memory-Mapped Files

```
>>> with open("v:/workspace/Python4_Lesson15/src/hello.txt", "wb") as f:
...     f.write(b"Hello Python!\n")
...
14
>>> import mmap
>>> with open("v:/workspace/Python4_Lesson15/src/hello.txt", "r+b") as f:
...     mapf = mmap.mmap(f.fileno(), 0)
...     print(mapf.readline())  # prints b"Hello Python!\n"
...     print(mapf[:5])  # prints b"Hello"
...     mapf.tell()
...     mapf[6:] = b" world!\n"
...     mapf.seek(0)
...     print(mapf.readline())  # prints b"Hello  world!\n"
...     mapf.close()
...
b'Hello Python!\n'
b'Hello'
14
b'Hello  world!\n'
>>>
```

The code above opens a file, then memory maps it. It exercises the readline() method of the mapped file, demonstrating that it works just as with a standard file. It then reads and writes slices of the mapped file (an equally valid way to access the mapped file's content, which does not alter the file pointer). Finally the file pointer is repositioned at the start and the (updated) contents are read in. (The "14" is the return value of the

write() function, which always returns the number of bytes written.)

OBSERVE:

```
>>> with open("v:/workspace/Python4_Lesson15/src/hello.txt", "wb") as f:
...     f.write(b"Hello Python!\n")
...
14
>>> with open("v:/workspace/Python4_Lesson15/src/hello.txt", "r+b") as f:
...     mapf = mmap.mmap(f.fileno(), 0)
...     print(mapf.readline())  # prints b"Hello Python!\n"
...     print(mapf[:5])  # prints b"Hello"
...     mapf.tell()
...     mapf[6:] = b" world!\n"
...     mapf.seek(0)
...     print(mapf.readline())  # prints b"Hello  world!\n"
...     # close the map
...     mapf.close()
...
b'Hello Python!\n'
b'Hello'
14
b'Hello  world!\n'
>>>
```

As we observed in an earlier lesson, **file objects are context managers**, albeit of a slightly degenerate kind (because they return themselves as the result of their __enter__() method). The first argument to **mmap.mmap** is a *file number* (an internal number used to identify the file to the operating system), which is obtained by **calling the file's fileno() method**. The **call to readline()** demonstrates normal file handling, but then you see **indexed access to the content**, which nevertheless demonstrates that **the file pointer is unchanged** by such access.

Next you see that the content of the file can also be changed **by subscripting**, though in this case it is essential that the new content is the same length as the slice being assigned. Finally you observed that the file had been changed by restarting at the beginning.

The difference between using a memory-mapped file and a standard one is that standard files are independently buffered in each process that uses them, meaning that a write to a file from one program is not necessarily immediately written to disk, and will not necessarily be seen immediately by a separate program reading the file using its own buffers.

The mmap Interface

For calls to **mmap.mmap()** to be cross-platform compatible they should stick to the following signature:

OBSERVE:

```
mmap(fileno, length, access=ACCESS_WRITE, offset=0)
```

The **file number** is used simply because this mirrors the interface of the underlying C library (not always the best design decision, but fortunately the file number is easily obtained from an open file's fileno() method). Using a file number of -1 creates an anonymous share (one that cannot be accessed from the filestore).

The call above maps **length** bytes from the beginning of the file, and returns an mmap object that gives both file- and index-based access to that portion of the file's contents. If **length** exceeds the current length of the file, the file is extended to the new length before operations continue. If **length** is zero, the mmap object will map the current length of the file, which in turn sets the maximum valid index that can be used.

The optional **access** argument can take one of three values, all defined in the mmap module:

Access Value	Meaning
ACCESS_READ	Any attempt to assign to the memory map raises a TypeError exception.
ACCESS_WRITE	Assignments to the map affect both the map's content and the underlying file.
ACCESS_COPY	Assignments to the memory map change the map's contents but do not update the file on which the map was based (a copy-on-write mapping).

The **offset** argument, when present, establishes an offset within the file for the starting position of the memory map. The offset must be a multiple of the constant **mmap.ALLOCATIONGRANULARITY** (which is typically the size of a virtual memory block, 4096 bytes on many systems).

What Use is mmap(), and How Does it Work?

The real benefit of mmap over other techniques is twofold: first, *the file is mapped directly into memory* (hence the name). When only one process is using the mapped file, this is a pedestrian application, but remember that modern computers use *virtual memory systems*. Each process's memory consists of a list of "memory pages." The actual address of the memory page does not matter to the process: the process accesses "virtual memory," and the hardware uses a "memory map" to determine whereabouts in a process's memory a particular page appears.

When a file is memory-mapped, the operating system effectively reserves enough memory to hold the whole file's contents (or that portion of the file that is being mapped) in memory, and then *maps that memory into the process's address space*. If another process comes along and maps the same file, then *exactly the same block of memory is mapped into the second process's address space*. This allows the two processes to exchange information extremely rapidly by writing into the shared memory. Since each is writing into the same memory, each can see the other's changes immediately.

Note

Be careful with large files. Remember that if you memory map a file it gets mapped into your process's virtual address space. If you are using 32-bit Python (either because you are running on a 32-bit system or because your system administrators chose to install a 32-bit Python interpreter on a system built using 64-bit technology), each process has a 4GB upper limit on the size of its address space. Since there are many other claims on a process's memory, it is unlikely you will be able to map all of a file much above 1GB in size in a 32-bit Python environment.

A Memory-Mapped Example

The following example code gives you some idea how memory-mapped files might be used for interprocess communication. The program creates a file that will hold data (encoded by the struct module) to be passed between the main program and its subprocesses. The file is split up into "slots," each large enough to hold a byte used to indicate the status of the slot, a 7-character string, and three double-length floating-point numbers. The status starts as **EMPTY**, and is set to the slot number every time new data becomes available. When there is no more data, the status is set to TERM, which indicates to the subprocess that there is no more work available.

The whole program is given in the listing below. This is a rather larger program than we normally ask you to enter in one go, but by now you should be able to understand what a lot of the code does as you type it in (explanations follow the listing).

Enter the following code as mpmmap.py

```
"""
mpmmap.py: use memory-mapped file as an interprocess communication area
           to support multi-processed applications.
"""

import struct
import mmap
import multiprocessing as mp
import os
import time
import sys

FILENAME = "mappedfile"
SLOTFMT = b"B7s3d"
SLOTSIZE = struct.calcsize(SLOTFMT)
SLOTS = 6 # Number of subprocesses
EMPTY = 255
TERM = 254

def unpackslot(byte_data):
    """Return slot data as (slot#, string, float, float, float)."""
    return struct.unpack(SLOTFMT, byte_data)

def packslot(slot, s, f1, f2, f3):
    """Generate slot string from individual data elements."""
    return struct.pack(SLOTFMT, slot, s, f1, f2, f3)

def run(slot):
    """Implements the independent processes that will consume the data."""
    offset = SLOTSIZE*slot
    print("Process", slot, "running")
    sys.stdout.flush()
    f = open(FILENAME, "r+b")
    mapf = mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_WRITE)
    while True:
        while mapf[offset] == EMPTY:
            time.sleep(0.01)
        if mapf[offset] == TERM:
            print("Process", slot, "done")
            sys.stdout.flush()
            mapf.close()
            return
        x, s, f1, f2, f3 = unpackslot(mapf[offset:offset+SLOTSIZE])
        print(x, slot, ":", s, f1*f2*f3)
        sys.stdout.flush()
        mapf[offset] = EMPTY

def numbers():
    """Generator: 0.01, 0.02, 0.03, 0.04, 0.05, ..."""
    i = 1
    while True:
        yield i/100.0
        i += 1

if __name__ == "__main__":
    f = open(FILENAME, "wb")
    f.write(SLOTSIZE*SLOTS*b'\0')
    f.close()
    f = open(FILENAME, "r+b")
    mapf = mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_WRITE)

    ptbl = []
    for slot in range(SLOTS):
        offset = slot*SLOTSIZE
        mapf[offset] = EMPTY
        p = mp.Process(target=run, args=(slot, ))
```

```
        ptbl.append(p)
        print("Starting", p)
        p.start()

    numseq = numbers()
    b = next(numseq)
    c = next(numseq)
    for i in range(4):
        for slot in range(SLOTS):
            a, b, c = b, c, next(numseq)
            offset = slot*SLOTSIZE
            while mapf[offset] != EMPTY:
                time.sleep(0.01)
            mapf[offset+1:offset+SLOTSIZE] = packslot(slot, b"*******", a, b, c)[1:]
            mapf[offset] = slot

    for slot in range(SLOTS):
        offset = SLOTSIZE*slot
        while mapf[offset] != EMPTY:
            time.sleep(0.01)
        mapf[offset] = TERM

    for p in ptbl:
        p.join()

    mapf.close()
    print(f.read())
    sys.stdout.flush()
    f.close()
    os.unlink(FILENAME)
```

There are a couple of utility functions for packing and unpacking the slot data; these are simple calls to standard **struct** functions that you may remember. Next comes the **run()** function that will be the meat of the subprocesses. When we call it, we pass the process's slot number, and it uses the computed size of the slot to work out where its particular portion of the data file begins. It then establishes a mapping onto the standard data file and goes into an infinite loop (which will be terminated by the logic it contains). It repeatedly looks at the first byte of its slot, until the EMPTY value it starts with is changed (by the main program). The process sleeps between different looks at the first byte, to avoid using too much CPU. The sleep should be long enough that the computations in the loop take a relatively insignificant time. If the value has changed to TERM, the process closes everything down and terminates. Otherwise it extracts the data from the slot, performs a calculation and prints out the results, and then sets the slot indicator back to EMPTY so the main program will refill the slot.

The run() function is followed by a simple numbers() generator function that separates the task of generating numbers from their use inside the main program. It is an infinite generator that yields numbers starting at 0.01 and increasing by 0.01 each call.

Now, we see the logic of the main program. The program first creates a data file large enough to contain the mapped data for all slots, then maps the file into memory. It then iterates over the slots, setting their status to EMPTY, creates a new process with the current slot number, saves it in a list and starts it. The newly-started process will wait until its slot is switched from EMPTY status before taking any action.

Next the program loops four times over all the slots, filling them with data and only then setting the slot indicator to the slot number. This avoids a potential hazard which might occur if the slot status was set at the same time as the rest of the data: it is just possible that a subprocess might see its status change and start trying to act before the rest of the data is copied in. Yes, this would be a low-probability occurrence, but that does not mean you are at liberty to ignore it.

Once the main loop is over, the program waits for each slot to become EMPTY and sets it to TERM to indicate that the associated process should terminate. Finally, the program waits for all the processes it started to terminate, deletes the file it created at the start of the run, and itself terminates. When you run the program, you should see the following output.

Output from mpmmap.py

```
Starting <Process(Process-1, initial)>
Starting <Process(Process-2, initial)>
Starting <Process(Process-3, initial)>
Starting <Process(Process-4, initial)>
Starting <Process(Process-5, initial)>
Starting <Process(Process-6, initial)>
Process 0 running
0 0 : b'*******' 6e-06
Process 1 running
1 1 : b'*******' 2.3999999999999997e-05
Process 3 running
3 3 : b'*******' 0.00012
Process 2 running
2 2 : b'*******' 5.9999999999999995e-05
Process 5 running
5 5 : b'*******' 0.00033600000000000004
Process 4 running
4 4 : b'*******' 0.00021000000000000004
0 0 : b'*******' 0.0005040000000000001
0 0 : b'*******' 0.0027300000000000002
3 3 : b'*******' 0.00132
1 1 : b'*******' 0.00072
4 4 : b'*******' 0.001716
2 2 : b'*******' 0.00099
5 5 : b'*******' 0.002184
3 3 : b'*******' 0.004896
4 4 : b'*******' 0.005814000000000001
2 2 : b'*******' 0.00408
1 1 : b'*******' 0.00336
0 0 : b'*******' 0.007980000000000001
4 4 : b'*******' 0.0138
5 5 : b'*******' 0.006840000000000001
3 3 : b'*******' 0.012143999999999999
2 2 : b'*******' 0.010626
1 1 : b'*******' 0.00924
5 5 : b'*******' 0.0156
Process 0 done
Process 4 done
Process 3 done
Process 1 done
Process 5 done
Process 2 done
b'\xfe*******R\xb8\x1e\x85\xebQ\xc8?\x9a\x99\x99\x99\x99\x99\xc9?\xe1z\x14\xaeG\xe1\xca
?\xfe*******\x9a\x99\x99\x99\x99\x99\xc9?\xe1z\x14\xaeG\xe1\xca?)\\\x8f\xc2\xf5(\xcc?\x
fe*******\xe1z\x14\xaeG\xe1\xca?)\\\x8f\xc2\xf5(\xcc?q=\n\xd7\xa3p\xcd?\xfe*******)\\\x
8f\xc2\xf5(\xcc?q=\n\xd7\xa3p\xcd?\xb8\x1e\x85\xebQ\xb8\xce?\xfe*******q=\n\xd7\xa3p\xc
d?\xb8\x1e\x85\xebQ\xb8\xce?\x00\x00\x00\x00\x00\x00\xd0?\xfe*******\xb8\x1e\x85\xebQ\x
b8\xce?\x00\x00\x00\x00\x00\x00\xd0?\xa4p=\n\xd7\xa3\xd0?'
```

Note The program above is for demonstration purposes only so you can start to understand the advantages of shared memory. The **multiprocessing** module actually has other ways to keep processes synchronized, and you should investigate those for production purposes. But if you understand the logic of the code above, you know what mapped files do and how they work, which is a significant piece of learning.

Memory-mapped files allow you to treat huge tracts of data as though they were large strings, and also allow you to share those large chunks of data between independent processes. They allow you to use inter-process communication.

In the final lesson, we consider some of the differences between small projects and large ones.

When you finish the lesson, don't forget to complete the homework!

Your Future with Python

Lesson Objectives

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

- find cool Python Conferences.
- explore the Python job market and career choices.
- explore new developments in Python.
- access a few new Python tips and tricks.

Python Conferences

Python is a rapidly growing language that attracts programmers all over the world. In the early 1990s an International Python Conference was started, which became the principal forum for live discussion of Python's uses and development (naturally extensive discussions were also held online, as they continue to be—but face-to-face meetings are still incredibly useful, and usually more productive than mailing list dicussions).

In 2002, I was asked by Guido van Rossum to chair a new type of conference, one that involved the community members far more, and was priced to allow those who didn't have professional budgets to come along and contribute. That first PyCon, in March 2003, attracted over 250 people, and established community conferences as the preferred mechanism for meeting up with other Python users (following in the footsteps of EuroPython, which had been held in Gotheborg, Sweden, a couple of months before). I chaired the first three conferences (by which time attendance had swollen to 450) and then handed the torch to Andrew Kuchling.

At the same time, PyCons were growing up in other countries, whose Python enthusiasts started to run national PyCons, and other, smaller, conferences are now held regionally in the USA (the first three of these were PyOhio, PyTexas and PyArkansas). PyCon Ireland, Kiwi PyCon, Python Brasil, PyCon AR (Argentina), PyCon UK, PyCon Italy, and many others. The Asia Pacific region recently started a pan-Asian conference (PyCon Asia Pacific) to support Python users in that region. It seems as though soon it will be impossible to avoid Python events clashing with each other simply because there are so many in the worldwide calendar.

In 2011 PyCon had 1200 delegates, and it currently looks set to start capping growth some time in the next two to three years (there is a general feeling that over-large conventions run the risk of losing the "community" flavor that is such an important part of conferences like PyCon). PyCon even runs a financial assistance scheme that regularly helps people who would otherwise not be able to afford to travel and attend PyCon. To learn more about these conferences, the best starting point is the root PyCon web site.

There are also a growing number of local user groups throughout the world. Some such groups use the popular MeetUp system to organize their groups, as it allows people to easily sign up for meetings and allows the meeting administrators reasonable control over attendance and the like.

All these activities are, in essence, run by members of the community (though certainly the larger ones like PyCon US are assisted by professional conference companies: volunteers cannot have their depth of experience, and must often make their contributions outside office hours.

Tutorials

As a conference matures and the organizers acquire more experience, you will often see tutorials offered at very reasonable prices. World authorities on various aspects of Python offer tutorials to help the Python Software Foundation to raise funds through the conference.

These tutorials are an amazing bargain, and an incredible way to learn new Python skills and techniques. Many of them cannot be taken anywhere else, and would alone be worth the price of conference registration.

Talks

The talks are the "meat" of most conferences, and Python conferences are no exception. Any given conference might include papers for beginners about some more obscure aspect of the language, intermediate papers on applications, or advanced stuff on how a particular framework achieves a certain effect using aspects of Python to achieve high efficiency (or other desirable aspects of their case).

Talks will typically be thirty minutes to an hour long, including time for questions. A lot of conferences are now putting out live video streams as the talks are presented (though with more than a couple of independent

tracks this can get rather demanding in bandwidth). The same video stream will be recorded, and there is a huge amount of Python-related material saved and available on the web. The primary searchable resource is the Python Miro Community, which tries to organize and index the material.

The Hallway Track

Much favored by experienced conference-goers, the hallway track is the discussions that take place between people outside the meeting rooms where talks are given. These discussions often arise completely spontaneously, but give better value than the rest of the conference. Even if you are new to conference-going, you should definitely keep your schedule open enough to take in the hallway track. And don't be surprised if some random conversation leads you to abandon your plans and use the hallway track instead.

Open Space

Many conferences now set aside space for participants to use for activities of their own choice. There are particular rules traditionally associated with the term "open space," but sometimes (to the annoyance of purists) the Python community simply interprets it as "rooms you can use for pretty much any conference-related activity." It is not unusual for speakers to invite interested audience members to an open space session where their questions can be answered in a more participative framework. You can get to meet some anazing people in open space .

Lightning Talks

Often the most entertaining sessions of the whole conference, the lightning talk sessions use five-minute slots in which speakers, who can often only sign up in person at the conference, must complete their presentation within the slot or be cut short by the session chairman.

If you are interested in becoming a conference speaker, presenting a lightning talk is a good way to dip a toe in the water. Audiences are very forgiving to new speakers and those who are not presenting in their first language. Topics are often light-hearted (one I particularly remember was "How I replaced myself with a small Python script"), and quite often introduce you to novel technologies that you would otherwise not have come across. Because the talks are short, the sessions go by quickly, and every speaker gets a round of applause.

Birds of a Feather Sessions (BOFs)

These are usually evening sessions, not formally organized but often using rooms in the conference venue, where people with a common interest in one specific area (testing, Django, numerical computing, Twisted networking, ...) get together and just share information in any suitable way. The Testing BOF has become a tradition on Saturday night at PyCon US, and runs lightning talks all of its own. In 2011 Testing BOF speakers were required to wear a white lab coat.

Sprints: Moving Ahead

Conferences are often followed by sprints—focused efforts on getting some aspect of a project up and running, by a team that might be scattered around the world when not actually at the same conference.

Sprints are a great place to learn about existing code bases: you can often get to talk with and learn from the people who wrote and/or are maintaining the code. Once you have met some of the people who contribute to the development, it is far less intimidating to join in and become a contributor yourself. The open source world only exists because people like us roll up their sleeves and start building things.

Whether local, national, or regional, Python conferences are an amazing way to improve your Python knowledge and increase your skill level. They are social as well as technical events, and when you become a regular conference-goer you will doubtless find, as do I, that there are people you look forward to meeting again and again, even if you only ever meet them at conferences.

The Python Job Market and Career Choices

Python is employed in such diverse ways, it is hard to think of an area of life that isn't affected by it one way or another. Google is well-known as an organization where Python is used extensively. Many organizations, including most of the USA's 100 largest newspapers, use a Python-based web framework called Django to build their web sites and maintain journalistic content.

In the scientific and engineering world, Python is equally versatile. The SciPy and Numpy packages put blazingly fast calculations and publication-quality graphics into the hands of scientists. This is done by using Python as a "glue" language to hold together high-speed logic written in compiled languages like Fortran and C, with most of the computation taking place in the compiled code.

Note The PyPy Python project is now reliably producing benchmark results that are several times faster than those of the CPython interpreter, although at present only available for Python 2.7. If this progress continues, Python could become a viable language in which to write numerical algorithms!

If you enjoy programming and want to carry on doing it, you will probably always find something to do. Programming is a great career if you like to find out about how things are (or can be) done. Of course, for many people programming will only be a part of their job, but that does not mean they can't enjoy it. Python can be used in so-called "embedded devices," the computers that are increasingly built into other equipment to act as a controlling element. Technicians of all kinds will find themselves thrust into programming as a part of their jobs, and having the introduction to Python that this Certificate Series has produced is a great introduction to programming generally (if you can program in Python, it is much easier to learn other languages).

If you want to know what jobs are currently available, the Internet is as usual your friend. The Python Software Foundation maintains a Jobs Board on which employers post jobs. Track that page for a while to get an idea of the range of jobs likely to be available, but many employers never find out about the Jobs Board. How do you find the other jobs? Well, the conventional ways all apply. For example, you can go to job search sites and enter "Python" as a keyword. You will find that many large companies are looking for Python skills.

In fact, as these words are being written, there is a worldwide shortage of Python skills. Clearly there is no guarantee how long this situation will last, but as long as it does, even fairly new programmers should be able to find jobs.

The difference between you as an O'Reilly School student and other applicants is that you have, over the course of your studies, been required to demonstrate understanding of the material and practical skills in applying it. You can show people code you have written, and can prove that you understand it and can talk sensibly about its structure. Even if you have not been studying for vocational reasons, we hope that you have found these methods helpful; if you're looking for work, they will set you apart from the average applicant. I have had to hire people, and it can be horrifying how many applications come from candidates who are obviously ill-qualified for the role or have only the shakiest grasp of programming concepts. So emphasize your practical experience: employers should regard it as valuable.

Python Development

This lesson is not intended to recruit Python core developers, but I am quite happy to encourage people with a sense of adventure to consider becoming one. Some beginners feel that they will not be wanted, or that their efforts will be unappreciated. This can seem so if the new developer's contributions are not reviewed sensitively, but people being people, this does not necessarily always happen.

Although not all developer-specific, most of the lists mentioned on the Python Mailing Lists page are concerned with some aspect of Python development or applications. The one exception is the general "comp.lang.python" list, which is a broad church in which you might expect to find anything from using a C debugger to whether Schrodinger's cat really does exist in two parallel states. It is fairly eclectic, and threads can ramble all over the place.

There is a new core-mentorship mailing list started specifically so that those with an interest in becoming a developer could interact with a gentler group than the whole developers list, and get a more welcoming reception. Once they have been inducted into the necessary processes, they are introduced to the rest of the developer community. Introductions are easier once someone has made an initial contribution.

Do not make the mistake of assuming that because the CPython interpreter is written in C you have to know the C language before you become a core contributor. The standard library and its test suite have lots of code written entirely in Python, and it needs maintenance just like everything else. Your Python skills are needed if you want to join the open source community!

There is quite a bit of material intended to help and encourage the new or would-be Python developer, concentrated in the python.org site. Although there may be a steep learning curve, contributing to Python's development can give you awesome rewards in terms of self-respect, and will also earn you kudos in the open source world that should transfer into other areas too.

Tips and Tricks

There is no stored collection of tips and tricks for you to rummage in (well, there is, it's called "Google" and it's accessible on the web). *Our* tips and tricks have been passed on as you have proceeded through your course work, in email discussions with your mentor, and through the training materials you have used.

You may even by now have begun to develop some sense of what is and is not "Pythonic," which should have improved the quality of your code somewhat. The simple rules continue to apply: as you write, express your code in the simplest way you can. Code that is easy to write is easy to read, and code that is easy to read is easy to maintain.

Code that is easy to maintain saves money in the long run because computer costs nowadays tend to be dominated by the costs of the people to program and run the systems.

Thus, if you stick with what you have learned, you should be able to get Python to help you do pretty much whatever you want it to do. That practical skill is the added value behind these classes.

Congratulations! You've just finished the final lesson in the fourth course of our Python Certificate Series! How cool are you? We sincerely hope that you've enjoyed these courses, and that you're a confident Python programmer. You've earned it!

When you finish the lesson, don't forget to complete the homework!