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.

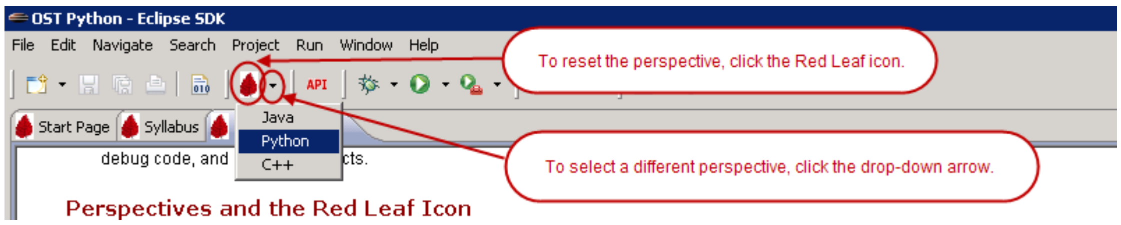
Perspectives and the Red Leaf Icon

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**:


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

To reset the perspective, click the Red Leaf icon.

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

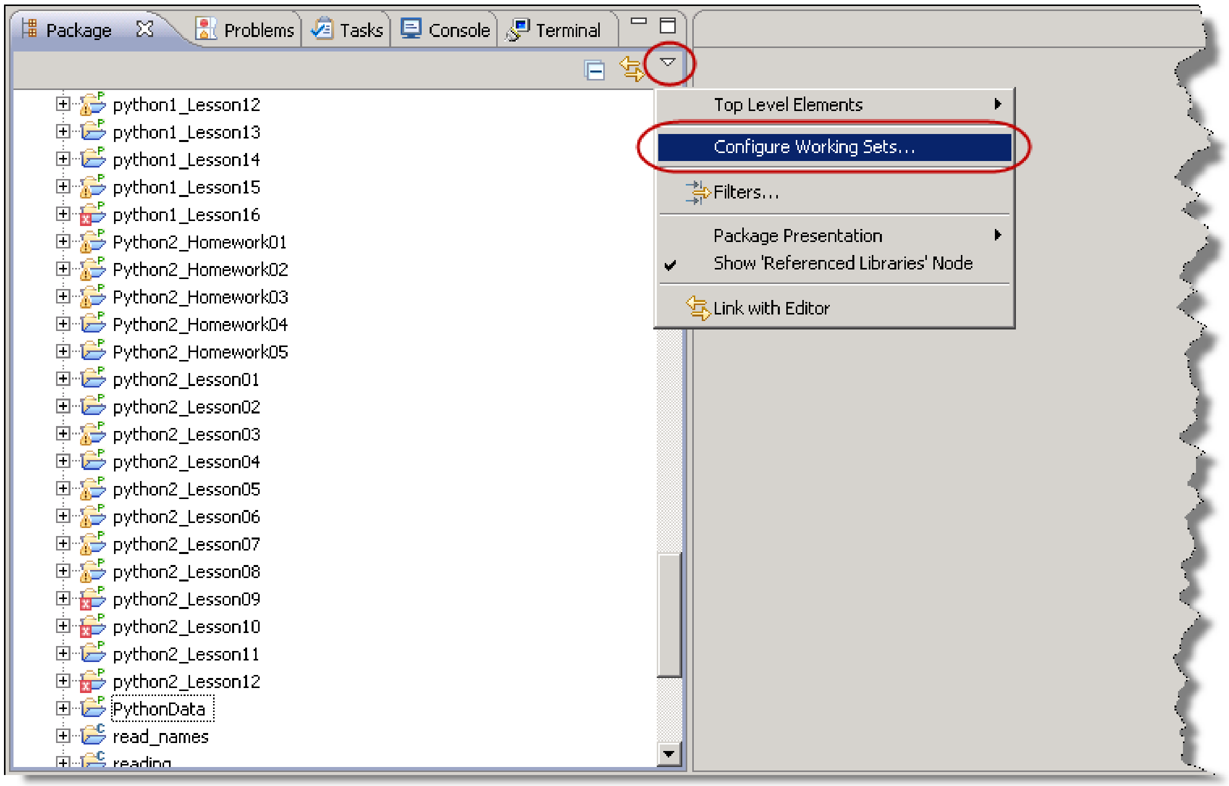
Working Sets

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.

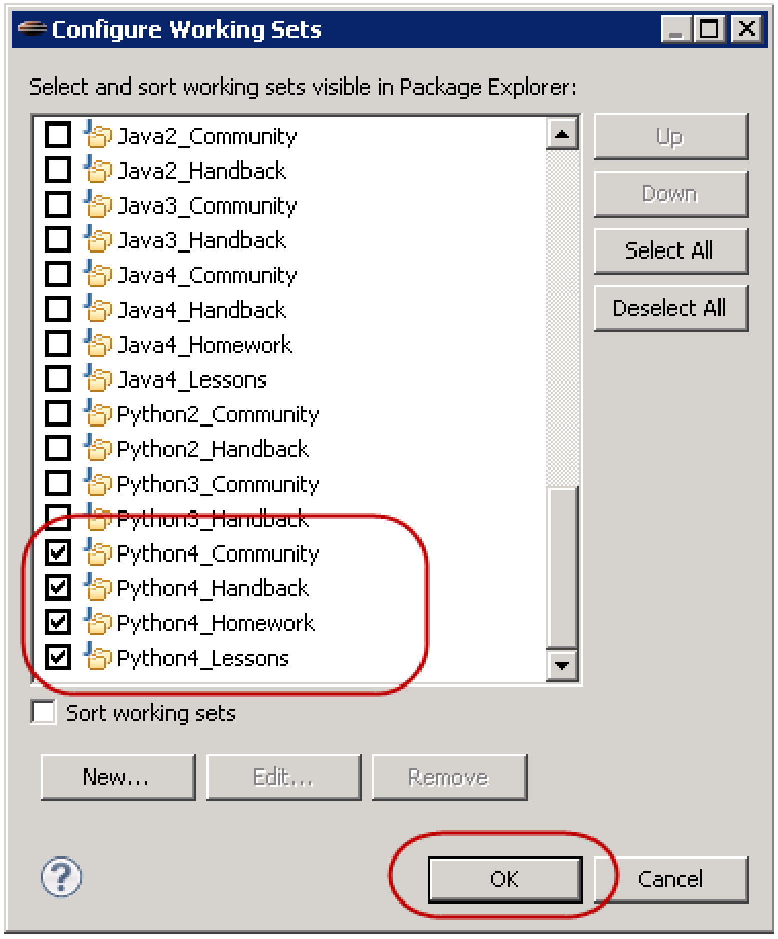
A working set is a convenient way to group related items together. You can assign a project to one or more working sets. In some cases, like the Python extension to Eclipse, new projects are created in a catch-all "Other Projects" working set. To organize your work better, we'll have you assign your 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.

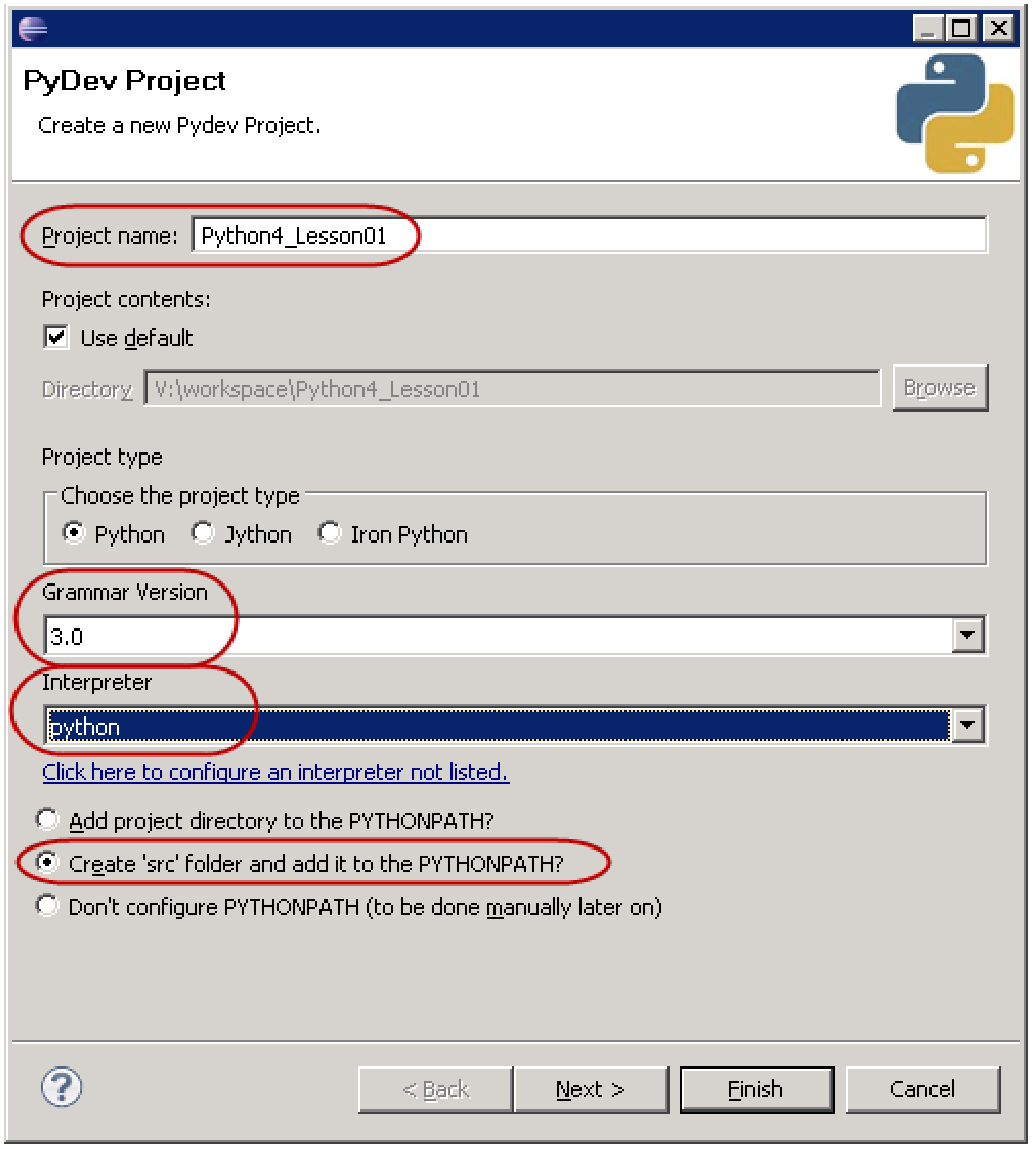
For this course, we'll display only the working sets you need. In the upper-right corner of the Package Explorer panel, click the downward arrow and select **Configure Working Sets**:



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:



PyDev Project

Create a new Pydev Project.

Project name: Python4_Lesson01

Project contents:

[✓] Use default

Directory V:\workspace\Python4_Lesson01

Project type

Choose the project type

● Python ○ Jython ○ Iron Python

Grammar Version: 3.0

Interpreter: python

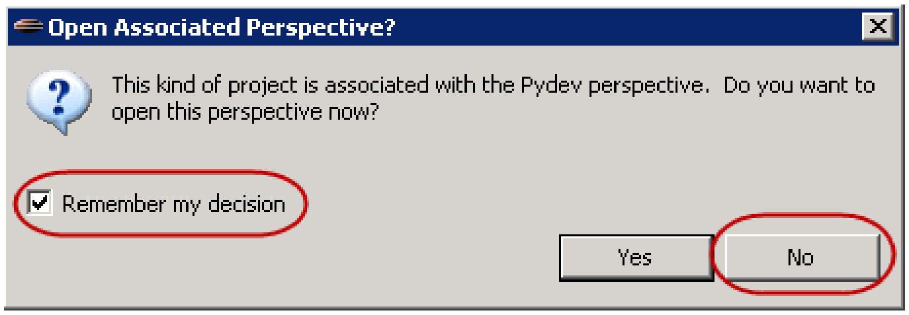
Click here to configure an interpreter not listed.

○ Add project directory to the PYTHONPATH?

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

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

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

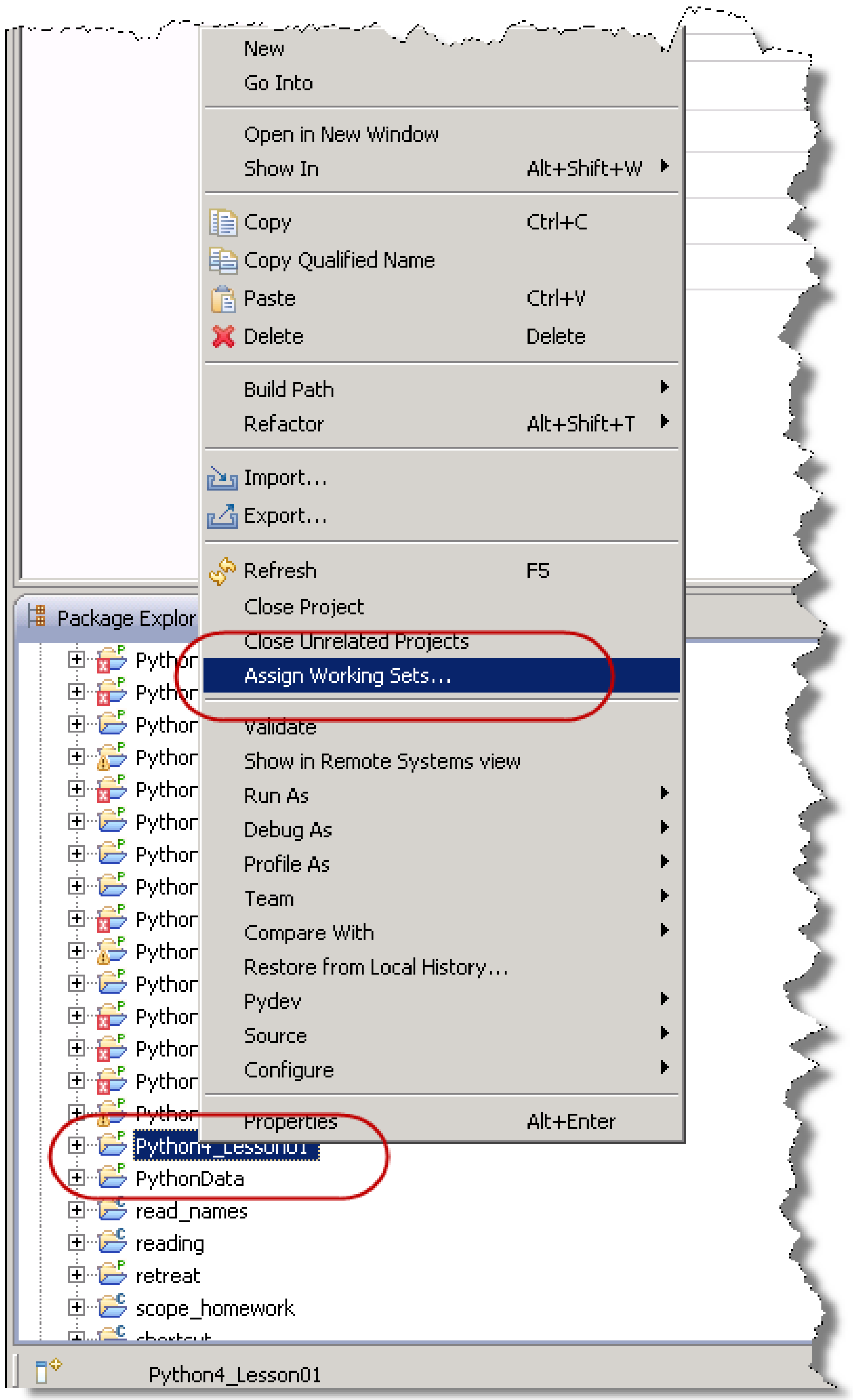


Open Associated Perspective?

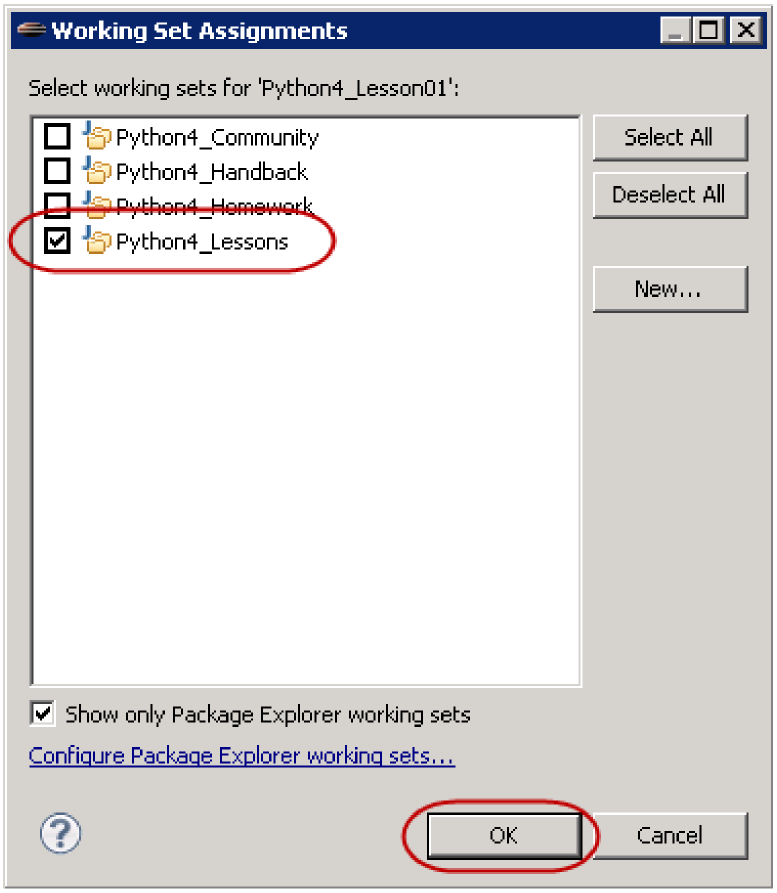
This kind of project is associated with the Pydev perspective. Do you want to open this perspective now?

[✓] Remember my decision

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



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



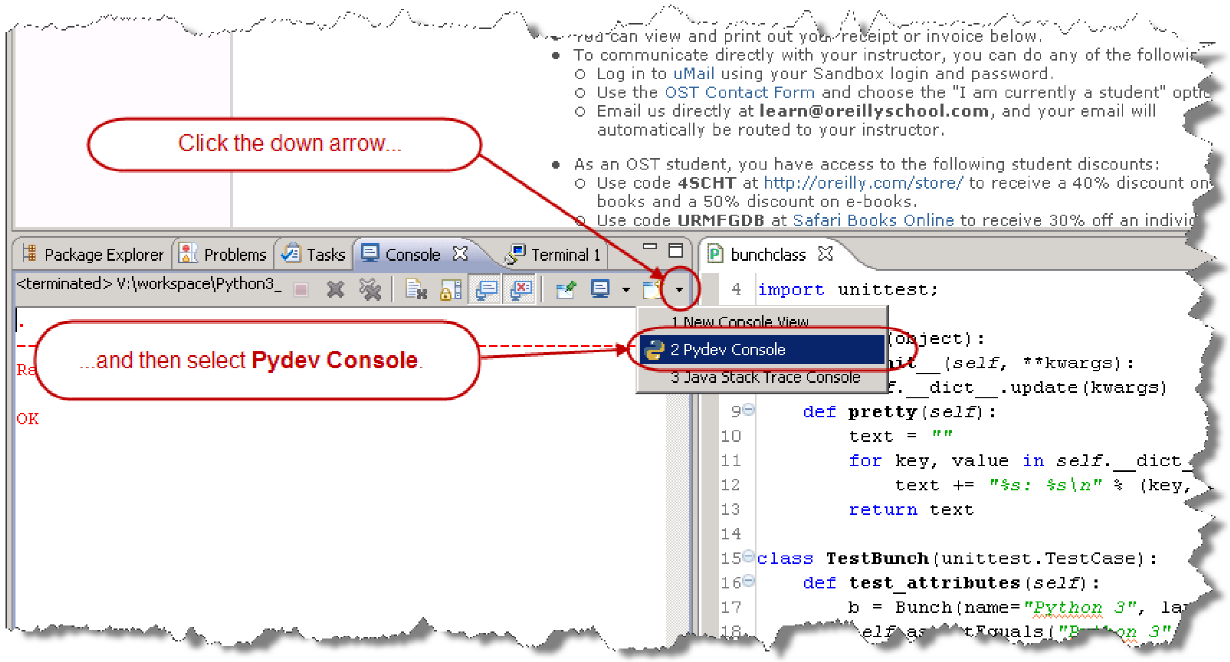
In the next section, we'll get to enter some Python code and run it!

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

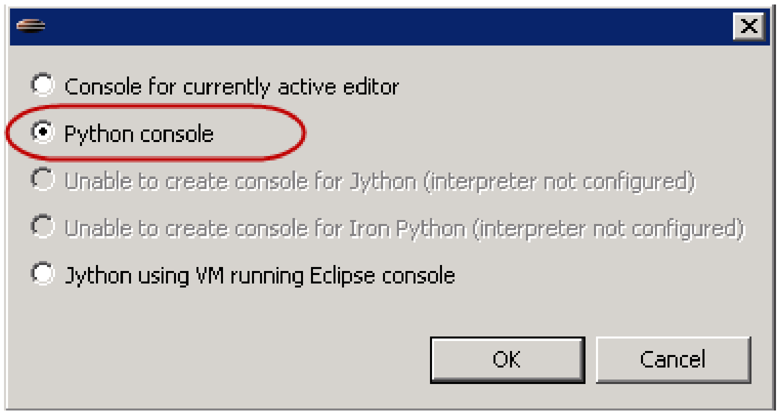
Like any object in Python, functions have a particular type; and like with any object in Python, you can examine a function's namespace with the **dir()** function. Let's open a new interactive session. Select the **Console** tab, click the down arrow and select **Pydev console**:



```
import unittest;
(object):
it__(self, **kwargs):
f.__dict__.update(kwargs)
def pretty(self):
    text = ""
    for key, value in self.__dict_
        text += "%s: %s\n" % (key,
    return text

class TestBunch(unittest.TestCase):
    def test_attributes(self):
        b = Bunch(name="Python 3", la
```

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



- ( ) Console for currently active editor
- [✓ Python console]
- ( ) Unable to create console for Jython (interpreter not configured)
- ( ) Unable to create console for Iron Python (interpreter not configured)
- ( ) Jython using VM running Eclipse 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 * square~~r~~
        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.

```
f = lambda x, y: x*y
        ≡
   def f(x, y):
         return x*y
```

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))

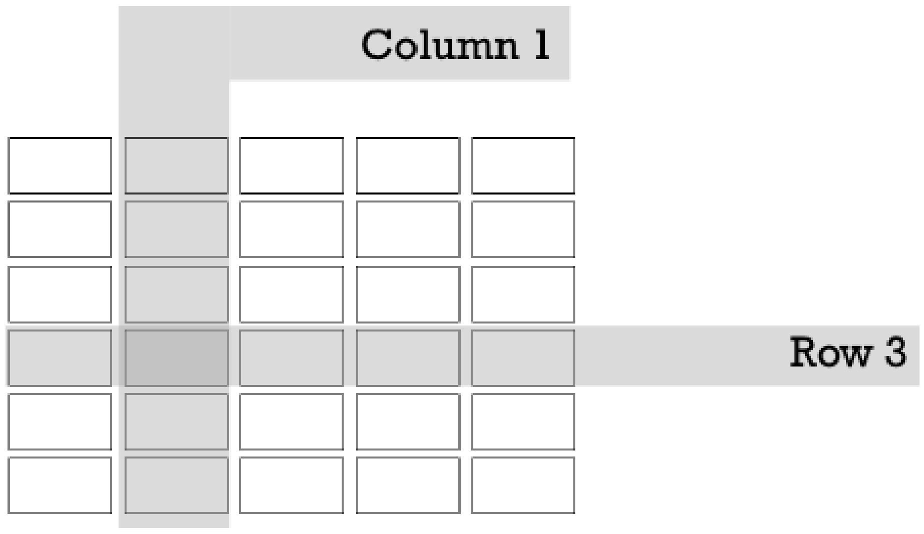
and

obj[M:N:S]

is equivalent to

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

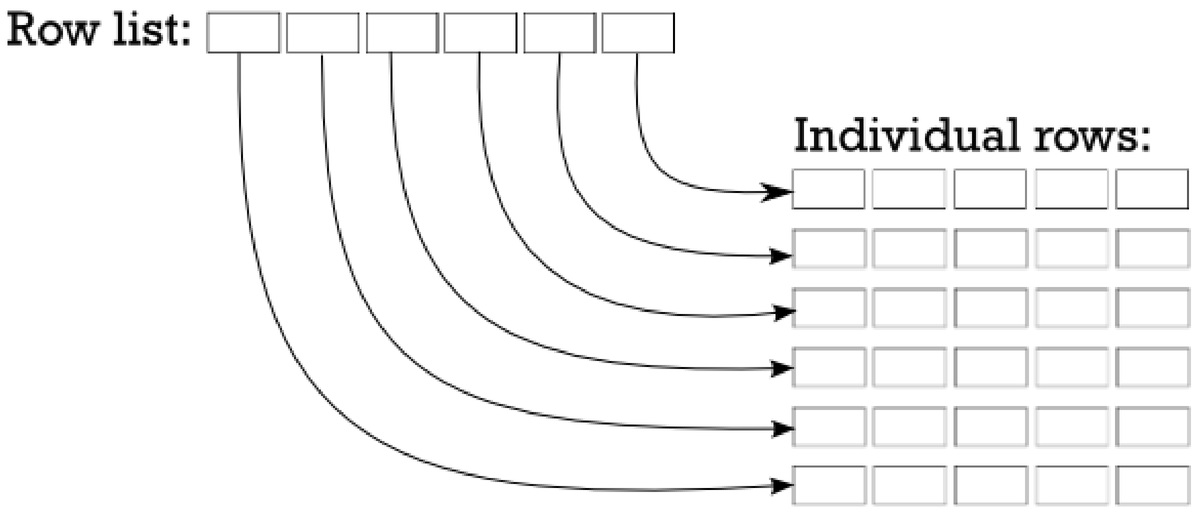
The list is a basic Python sequence, and like all the built-in sequence types, it is one-dimensional (that is, any item can be addressed with a single integer subscript of appropriate value). But multi-dimensional lists are often more convenient from a programmer's perspective, and, with the exception 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 * N~~sys_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()))
```

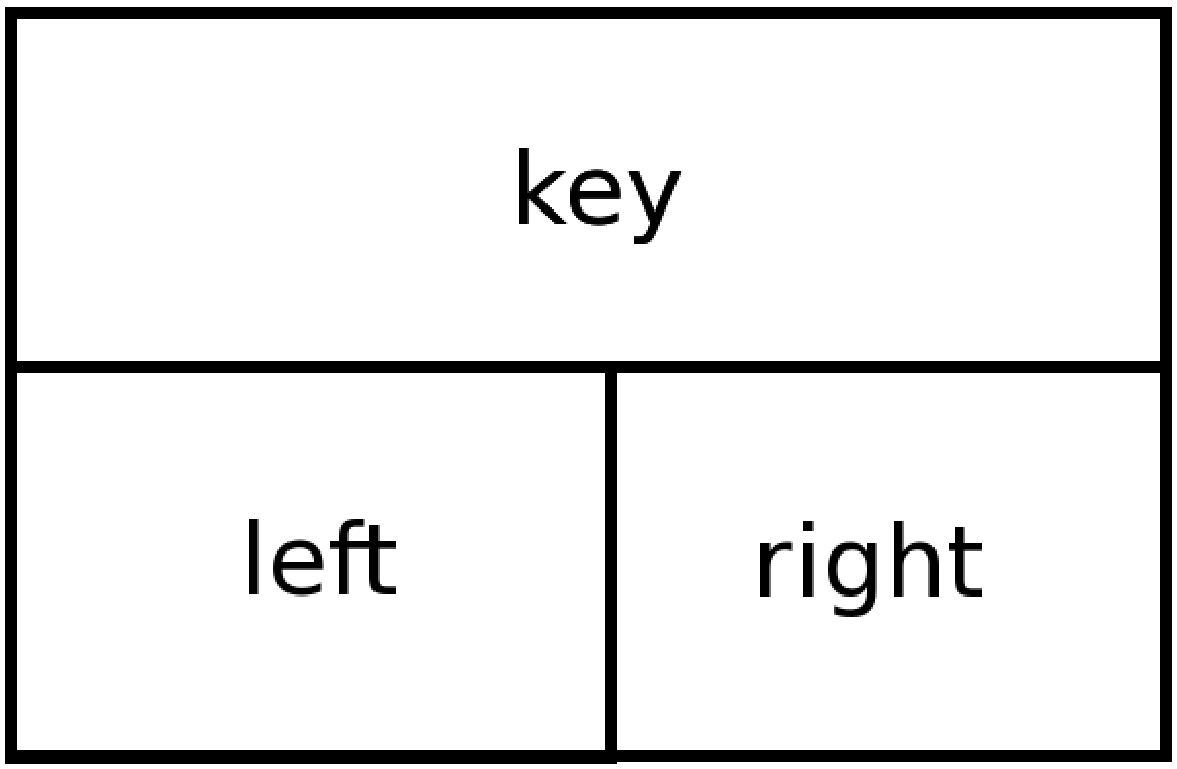
▫ Here again we chose not to have you write tests for your code, but we do test it rather informally with the code following the class declaration. The tree as created, consists of a single node. After creation, a loop inserts a number of characters, and then finally the **walk()** method is used to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

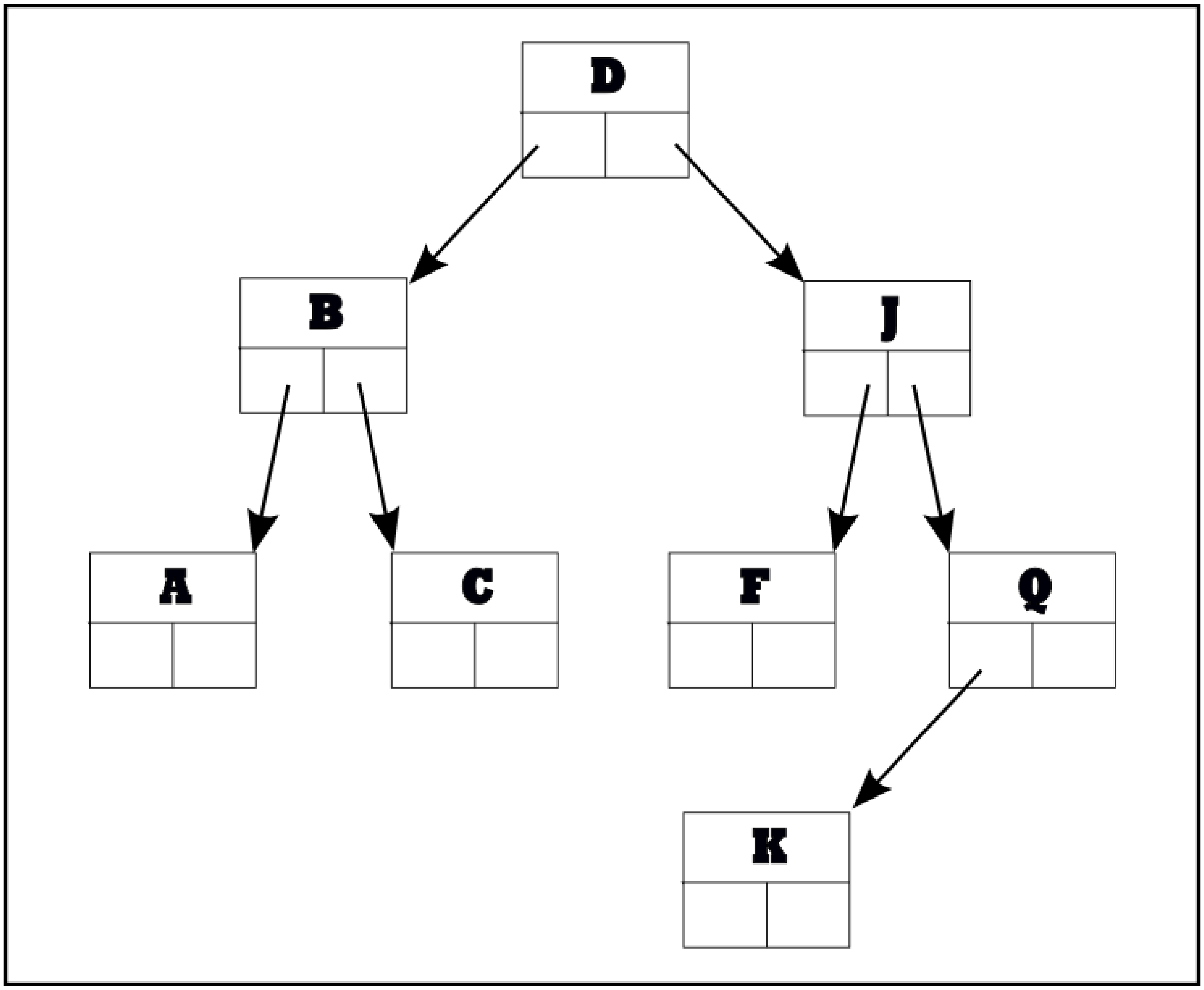


Such recursive algorithms aren't quite the same as delegation, but still, you could think of walk() and insert() as delegating a part of the processing to the subtrees. When you run **tree.py**, you'll see this:

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.