Chapter 4

Modules

CHAPTER OUTLINE

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00004-1

BY THE END OF THIS CHAPTER, YOU WILL BE ABLE TO:

- Describe what a module is.
- Explain how scope relates to modules.
- Differentiate global and local symbol tables.
- Define what attributes are in Python.
- Create your own module.
- Describe the default attributes of any module.
- Use comments to document your modules for use with the **help()** function.
- Compare and contrast packages and ordinary modules.
- Create your own packages to organize your tools.
- Compare and contrast **import** and **reload()**.
- Compare and contrast Python's **import** keyword and MEL's `source` directive.
- Use the **as** keyword to assign names to imported modules.
- Use the **from** keyword to import attributes into the current namespace.
- Temporarily modify Python's search path using the **sys.path** attribute.
- Create a userSetup script to automatically configure your environment.
- Configure the PYTHONPATH environment variable in Maya.env.
- Create complex deployments using systemwide environment variables.
- Describe the advantages of using a Python IDE for module development.
- Locate and install a Python IDE.

Thus far, we have used Maya's Script Editor to do all our work. The Script Editor is perfectly fine for testing ideas, but once your script is functional, you will want to save it as a module for later use. Although you have been *using* modules all along, this chapter will discuss everything you need to start *creating* modules.

We begin by discussing what exactly a module is and how modules manage and protect data in a Python environment. We then discuss how to

create modules as well as how to organize modules into packages. Next, we focus on exactly what happens when a module is imported and examine different options for importing modules. We then dive into a multitude of ways to configure your environment and deploy the Python tools to your studio. Finally, we examine a few different options you have for doing advanced, rapid Python development.

WHAT IS A MODULE?

A *module* is a standalone Python file containing a sequence of definitions and executable statements. Modules can contain as few or as many lines of code as are required, and often focus on specific tasks. Modules can also import other modules, allowing you to organize your code by functionality and reuse functionality across tools. While modules allow you to easily reuse code in this way, they are also designed to protect your code.

Modules are simply text documents with the .py extension. When you import a module for the first time, a corresponding .pyc file will be generated on-the-fly, which is a compiled module. A compiled module consists of what is called bytecode, rather than source code. While source code is human readable, bytecode has been converted into a compact format that is more efficient for the Python interpreter to execute. As such, you can deploy your modules as either .py or .pyc files. Many modules that ship with Maya's Python interpreter exist only as bytecode to protect their contents from accidental manipulation.

Python itself ships with a large library of default modules. There are all kinds of modules for various tasks, including doing math, working with emails, and interfacing with your operating system. Although we use several of these standard modules throughout this book, you will want to check out the online Python documentation at some point to view a full list of built-in modules and descriptions of their functions. Often, a module already exists that solves some problem you may be encountering!

In addition to these built-in modules, there are many useful modules available in the Python community at large. For example, the numpy package contains useful modules for scientific computing and linear algebra, while packages such as pymel offer powerful tools specifically designed to extend Maya's built-in cmds and API modules.

In the remainder of this chapter we will focus on creating custom modules. Before you create your first modules, however, it is worth reviewing the role of scope in modules.

MODULES AND SCOPE

We have briefly discussed the concept of scope when learning about functions. Scope is also important when working with modules. When you execute a line of code in the Script Editor, you are working in a particular scope. Consider the following example.

1. Enter the following line in the Script Editor to create a new variable.

```
character = 'Bob';
```

2. Python has two built-in functions, **globals()** and **locals()**, that allow you to see what objects exist in the current global and local scope, respectively. Execute the following line in the Script Editor to see the current global symbol table.

```
print(globals());
```

   If you just started a new Maya session, you should see a fairly short dictionary like that shown in the following output. If you have been executing other Python scripts in the Script Editor though, your output may be significantly longer.

```
{'__builtins__': <module '__builtin__' (built-in)>,
'__package__': None, '__name__': '__main__', '__doc__':
None, 'character': 'Bob'}
```

   In the dictionary, you can see the `character` variable you just defined, as well as its current value. However, a variable with such a name is quite generic. It is likely that you or another programmer will create another variable with the same name. If your program were to rely on this variable, then you may have a problem.

3. Assign a new value to the `character` variable and print the global symbol table again.

```
character = 1;
print(globals());
```

   As you can see in the output, Bob is no more, and the `character` variable now has a number value.

```
{'__builtins__': <module '__builtin__' (built-in)>,
'__package__': None, '__name__': '__main__', '__doc__':
None, 'character': 1}
```

If every script we created were storing variables in this scope, it would be very difficult to keep track of data. For example, if you were to execute two scripts one after the other, and if they contain variables with the same names, you may have conflicts. This problem can easily arise when working in MEL, necessitating that you devise exotic naming conventions for

variables. When using Python modules, however, you can avoid many such conflicts.

In contrast to the global symbol table, the local symbol table, accessible with the **locals()** function, lists the names of all symbols defined in the current scope, whatever that may be. If you simply execute **locals()** in the same way you have **globals()**, the difference may not be obvious, as the symbol tables are identical at the top level. However, the local scope may be inside a function.

4. Execute the following lines in the Script Editor.

```
def localExample():
    foo = '1';
    print(locals());
localExample();
```

As you can see, the local symbol table in the function contains only the variable that has been defined in its scope.

```
{'foo': '1'}
```

It is recommended that you consult Section 9.2 of the Python Tutorial online for more information on scope and how names are located.

Module Encapsulation and Attributes

Modules are designed to encapsulate data in their own symbol tables, eliminating the concern that you may overwrite data accidentally. *Each module has its own global symbol table*, which, as you have seen, you can access using the **globals()** function. By default, importing another module adds it to the current global symbol table, but its variables and other data are only accessible using the name assigned to it in the current namespace. Consider the following short sample.

```
import math;
print(math.pi);
```

The math module contains several such variables. *Variables and functions that are part of a module are called attributes*. (Unfortunately, Maya also uses the term *attributes* to describe properties of nodes, but you can hopefully follow.) To access an attribute in the math module—**pi** in this case—we supplied the name that was assigned to the imported module, a period, and then the name of the attribute. Follow the same pattern when accessing functions or other data in a module. As you have seen in previous chapters, the maya.cmds module contains a variety of functions for executing Maya commands.

In this case, the **pi** attribute is said to exist in the namespace of the math module. This concept ensures that any reference to the **pi** attribute must

be qualified by specifying its namespace, guarding against unintentional changes to its value. For example, you can create a new variable `pi` and it will not interfere with the one in the `math` namespace.

```
pi = 'apple';
print('our pi: %s'%pi);
print('math pi: %.5f'%math.pi);
```

The output clearly shows that the variables are referencing different data.

```
our pi: apple
math pi: 3.14159
```

In this example, we have a `pi` variable defined in the global symbol table, and the math module continues to have the correct value for its **pi** attribute in its global symbol table.

Since Python encapsulates the data in a module, we can safely import a module into another module. However, it is important to note that module attributes are not private in the sense used in other languages. It is possible to modify attribute values externally. Consider the following example.

```
pitemp = math.pi;
math.pi = 'custard';
print('override: %s'%math.pi);
math.pi = pitemp;
print('reassigned: %.5f'%math.pi);
```

As you can see from the output, it is possible to override the value of the **pi** attribute in the math module.

```
override: custard
reassigned: 3.14159
```

This aspect of Python is actually incredibly useful in some cases, as it allows you to override any attributes, including functions! For instance, you can override string output when printing API objects to the console, a trick of which the value will be clearer as you start working with the API. However, with great power comes great responsibility. In this case, we effectively "checked out" the existing value to another variable and "checked it back in" when we were done. It is often important to exercise good citizenship in this way, as other modules may rely on default values and behavior.

The __main__ Module

It is now worth pointing out that the Maya Script Editor is creating your data in a module as well. This module is called __main__. Every module has a few default attributes and one of these is called **__name__**, which you probably saw when you printed the global symbol table earlier. If

you print the **__name__** attribute in the Script Editor you will see the name of the current module.

```
print(__name__);
```

As you can see, the current module that you are in has the name of __main__. The Maya Script Editor is the root module and all other modules branch out from it. This concept is illustrated in Figure 4.1. In Maya, our scripts cannot be the __main__ module, as Maya is acting as the __main__ module.

It is worth briefly noting that any Python script executed as a shelf button (as we demonstrated in Chapter 1) is executed in the __main__ module, just as if it were executed in the Script Editor.

1. Type the following code in the Script Editor but do not execute it.

   ```
   character = ('Nathan', 'Chloe', 'Elena');
   ```

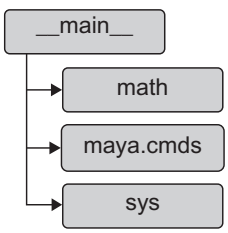
2. Highlight the code you just entered in the Script Editor and **MMB** drag the line to the Shelf. If you are using a version of Maya that asks, make sure you specify the code is written in Python.
3. Press the new custom button and you should see the new assignment output appear in the Script Editor's History Panel.
4. Execute the following line to print the global symbol table for __main__.

   ```
   print(globals());
   ```

   You should see from the output that the value of `character` has changed once more.

   ```
   {'__builtins__': <module '__builtin__' (built-in)>,
   '__package__': None, '__name__': '__main__', '__doc__':
   None, 'math': <module 'math' from '/Applications/
   Autodesk/maya2013/Maya.app/Contents/Frameworks/Python.
   framework/Versions/Current/lib/python2.6/lib-dynload/
   math.so'>, 'character': ('Nathan', 'Chloe', 'Elena')}
   ```

Scope is an essential concept not only of modules but also of Python more generally. We'll examine a few more aspects of scope throughout this text,

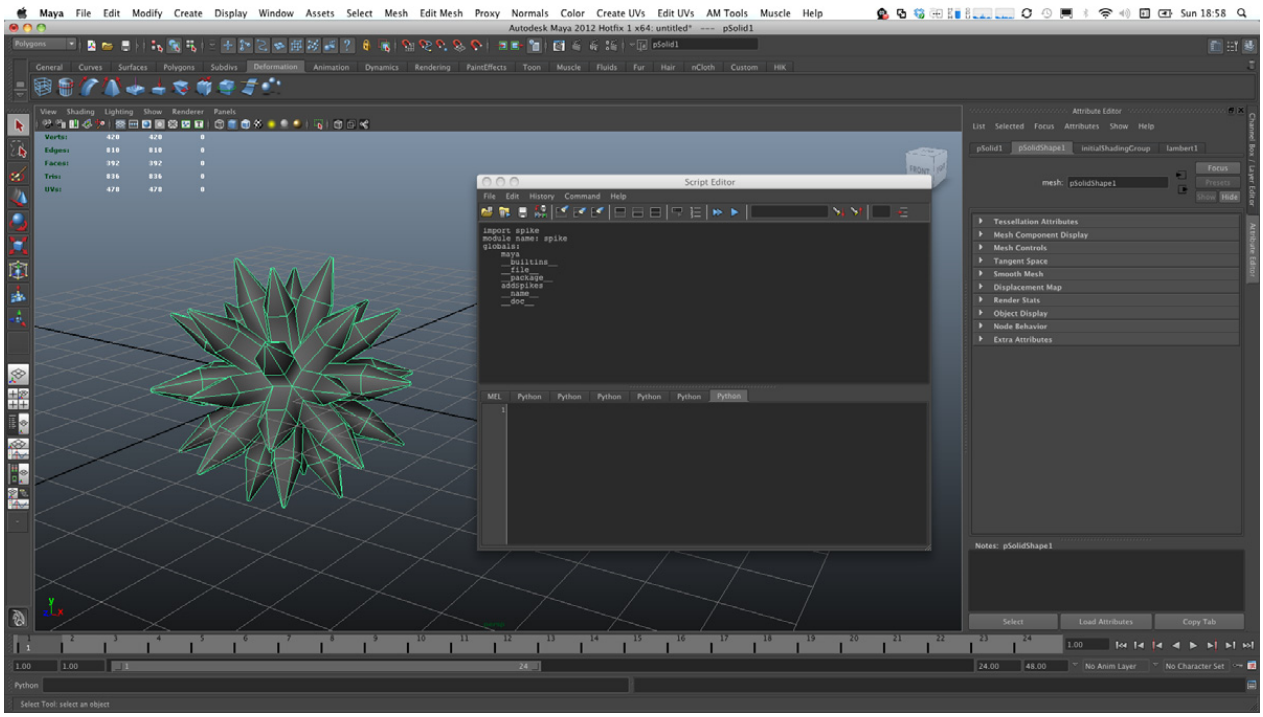


■ **FIGURE 4.1** Example hierarchy of three imported modules.

but it is always a good idea to consult the Python documentation for more information. At this point, we've covered enough basic information for you to start creating your own modules.

CREATING A MODULE

Creating your own module is simple, and only requires that you create your script in any text editor and save it with the .py extension. You can also save a script that you have entered in Maya's Script Editor directly from its menu. We will start by using the Script Editor to create our first module: spike. This module contains a function to add spikes to a polygon object, and will create a spiky ball model as shown in Figure 4.2.



■ **FIGURE 4.2** The results of the spike module.

The spike Module

```
"""This script prints some information and creates a spike
ball."""
import maya.cmds;
def addSpikes(obj):
    """This function adds spikes to a polygon object."""
    try: polycount = maya.cmds.polyEvaluate(obj, face=True);
```

```
    except: raise;
    for i in range(0, polycount):
        face = '%s.f[%s]'%(obj, i);
        maya.cmds.polyExtrudeFacet(face, ltz=1, ch=0);
        maya.cmds.polyExtrudeFacet(
            face, ltz=1, ch=0,
            ls=[0.1, 0.1, 0.1]
        );
    maya.cmds.polySoftEdge(obj, a=180, ch=0);
    maya.cmds.select(obj);
print('module name: %s'%__name__);
print('globals:');
for k in globals().keys(): print('\t%s'%k);
addSpikes(maya.cmds.polyPrimitive(ch=0)[0]);
```

1. Download the spike.py module from the companion web site or copy the preceding text into the Script Editor. If you have downloaded the script, then open it in Maya by selecting **File → Load Script** from the Script Editor window. The contents of the module should now appear in the Input Panel. Do not execute them.
2. Select **File → Save Script** from the Script Editor window. Name the script spike.py and save it to a location in your Python path based on your operating system:
   - **Windows:** C:\Documents and Settings\user\My Documents\maya\<version>\scripts\
   - **OS X:** ~/Library/Preferences/Autodesk/maya/<version>/scripts/
   - **Linux:** ~/maya/<version>/scripts/
3. In an empty Python tab, import the spike module and you should see some output in your History Panel.

```
import spike;
```

When you first imported the spike module, its contents were executed. If you look at the module's source, you can see that it defines a function for adding spikes to a polygon model, then prints some information about the module and executes the **addSpikes()** function, passing in a basic polygon ball. We included the printout information to provide a better picture of what is going on in the module.

```
module name: spike
globals:
    maya
    __builtins__
    __file__
    __package__
    addSpikes
    __name__
    __doc__
```

In addition to printing the name of the module, we printed its global symbol table using the **globals()** function. As you can see, this module's global symbol table differs from that of the __main__ module, which we investigated earlier. These names are all those existing in the module's namespace. As such, its data are separate from those defined in the __main__ module's namespace, which we investigated earlier. This example clearly demonstrates the encapsulation principle.

Default Attributes and help()

Looking at the global names in the spike module, you can see five attributes that have a double underscore at the beginning and end of the names. These five attributes will always be included with any module, even if they are not explicitly defined. We have already discussed **__name__**.

The **__builtins__** attribute points to a module that contains all of Python's built-in functions such as **print()** and **globals()**, and objects such as int and dict. This module is always automatically loaded, and does not require that you access its attributes using a namespace.

The **__file__** attribute of a module is simply a string indicating the absolute path to the module's .py file on the operating system.

The **__package__** attribute of a module returns the name of the package to which the module belongs. Because the spike module is not part of a package, this attribute returns a value of None. We will talk more about packages later in this chapter.

The **__doc__** attribute is very useful when used correctly. Python assumes the first comment in a module is its docstring. Likewise, each function has its own **__doc__** attribute, also presumed to be the first comment in its definition. If you properly add comments to a module in these locations, you can use the built-in Python **help()** function to get information.

```
help(spike);
```

We added comments in the spike module specifically to demonstrate this feature. Though many of our examples in this book exclude comments to save space, it is good practice to always include docstrings for your modules and functions.

```
Help on module spike:

NAME
    spike - This script prints some information and creates a
    spike ball.
```

```
FILE
    c:\documents and settings\ryan trowbridge\my documents
    \maya\2011\scripts\spike.py

FUNCTIONS
    addSpikes()

        This function adds spikes to a polygon object.

DATA
    k = '__doc__'
```

The output under the NAME heading shows the name of the module, which is identical to the value stored in its **__name__** attribute. The name of the module is followed by the comment at the beginning of the module, which specifies what the module is for. Similarly, the FILE section shows the path to the module's source file by using the value in the **__file__** attribute.

The FUNCTIONS section shows a list of all of the functions in the file, as well as their docstrings if they have them. Remember that function docstrings are specified just like module docstrings by using a literal string comment right at the beginning of the definition.

The final section, DATA, shows the noninternal attributes contained in the module. This section lists any module attributes that are not prefixed with underscores. In this example, we see the final value that was in the **k** attribute in our iterator to print the spike module's global symbol table.

Conveniently, you can also use the **help()** function with individual functions, which is useful with large modules like math. Consider the following example.

```
import math;
help(math.atan);
```

You should see output describing the **atan()** function.

```
Help on built-in function atan in module math:

atan(...)
    atan(x)

    Return the arc tangent (measured in radians) of x.
```

Packages

In addition to ordinary, individual modules, Python allows you to organize your modules into special hierarchical structures called *packages*. You can think of a package as essentially a folder that is treated as a module.

Packages allow you to cleanly deploy suites of tools, rather than requiring that end users fill up one folder with a bunch of modules, or that you create extraordinarily long modules. Packages provide both organization and technical advantages over ordinary modules. First, you are able to bypass naming conflicts by nesting modules inside of packages (or even packages inside of packages), reducing the likelihood of butting heads with other tools. Second, modules contained within a package do not need to have their disk locations explicitly added to your Python search path. Because the package folder itself is treated as a module, you can access modules in the package without having to append their specific locations to the search path. We talk more about the Python search path in the next section.

Packages are easy to create. The basic requirement is that you add a file, __init__.py, into a folder that is to be treated as a package (or a subpackage). That's it! After that point, you can then import the folder as though it were a module, or import any modules in the folder by using the dot delimiter. We'll walk through a brief example to demonstrate a couple of the features of packages.

1. Return to the location where you saved the spike.py module in the previous example and create a new folder called primitives. As an alternative to steps 1–3, you can also download the primitives package from the companion web site and unzip it here.
2. Create two files in this new folder called create.py and math.py and copy the contents from the examples shown here. The create module wraps some default Maya commands for creating polygon primitives, returning the **shape** nodes from the creation calls. The math module contains two simple utility functions for accessing attributes on primitives that have **radius** and **height** attributes.

create.py

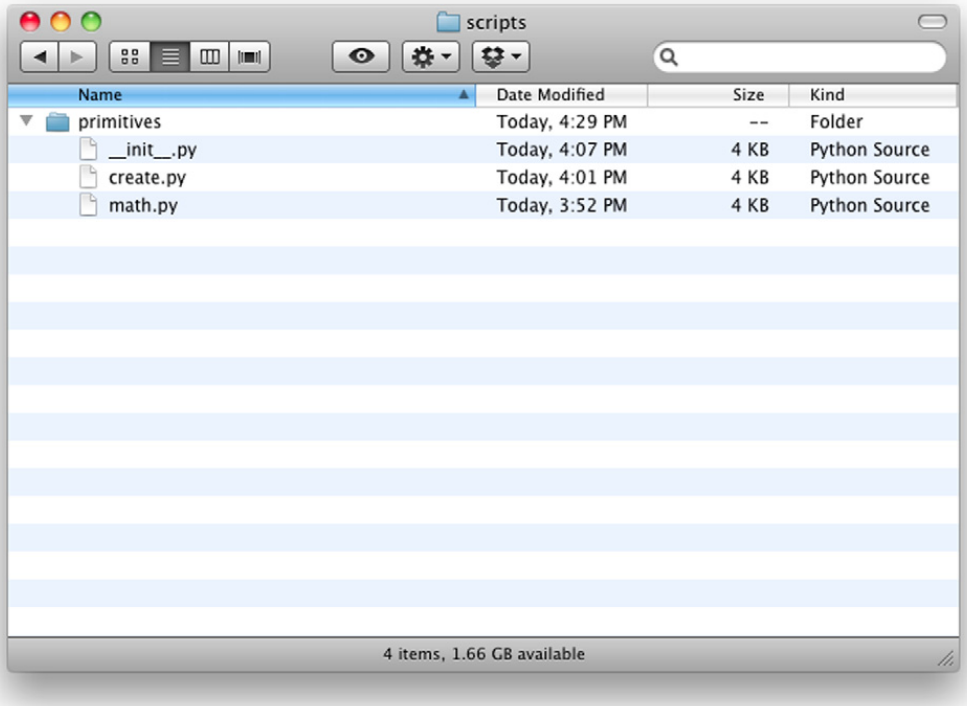
```
import maya.cmds;
def cone(**kwargs):
    try: return maya.cmds.polyCone(**kwargs)[1];
    except: raise;
def cube(**kwargs):
    try: return maya.cmds.polyCube(**kwargs)[1];
    except: raise;
def cylinder(**kwargs):
    try: return maya.cmds.polyCylinder(**kwargs)[1];
    except: raise;
def plane(**kwargs):
    try: return maya.cmds.polyPlane(**kwargs)[1];
    except: raise;
```

```
def sphere(**kwargs):
    try: return maya.cmds.polySphere(**kwargs)[1];
    except: raise;
def torus(**kwargs):
    try: return maya.cmds.polyTorus(**kwargs)[1];
    except: raise;
```

math.py

```
import maya.cmds;
pi = 3.1415926535897931;
def getCircumference(obj):
    try: return maya.cmds.getAttr('%s.radius'%obj) * 2.0 * pi;
    except: raise;
def getHeight(obj):
    try: return maya.cmds.getAttr('%s.height'%obj);
    except: raise;
```

3. Add another file in this new folder called __init__.py. This file turns the folder into a package. Copy the contents from our example here into this file. This file will simply import two modules in the package. You should now have a directory structure like that shown in Figure 4.3.



■ **FIGURE 4.3** Organization of the primitives package.

__init__.py

```
import create, math;
```

4. In Maya, you can now import the primitives package and use it to create objects and access information about the objects you create. Execute the following lines.

```
import primitives;
cyl = primitives.create.cylinder(r=0.25);
print(
    'Circumference is %.3f'%
    primitives.math.getCircumference(cyl)
);
```

You should see the circumference printed to three decimal places.

```
Circumference is 1.571
```

The first interesting point about this example is, as you noticed, we created a module called math. Because this module is nested in a package, its name can conflict with the built-in math module without our having to worry that we will affect other tools.

That being said, however, the second interesting point is that our custom math module *does* conflict in any of the namespaces in the primitives package. For instance, we had to add our own attribute **pi** to the math module, and could not import the **pi** attribute associated with the built-in math module. The same rule goes for importing the math module from __init__.py.

This point leads nicely to the third, which concerns how we imported the package's modules in the __init__.py file. We only need to supply the module names as they exist in the package, and do not need to reference them by nesting them, as in the following line.

```
import primitives.create;
```

When importing modules inside a package, modules in the package are considered to be in the search path, and will take precedence if there are any conflicts.

Finally, note that it is not necessary that the __init__.py file actually contain any code. This file simply indicates that the folder is a package. Adding these import calls, however, allows the modules in the package to be accessed as attributes without requiring their own separate imports. For example, by importing the package's modules in __init__, we are immediately able to access them as attributes of `primitives` in __main__. Without these import calls, we would need to separately import primitives .create and primitives.math in __main__.

Hopefully, the value of packages is clear from this simple example, as you will use them frequently in production. In practice, you will use packages to organize complex tools on a functionality basis. For example, you may put all of your project's tools into one package and have subpackages inside of it for animation, effects, modeling, rigging, utilities, and so on. While you are unlikely to be so flippant as to create modules of which the names conflict with those of built-in modules, packages do allow you a good deal of assurance that your tools are not likely to conflict with other tools your team members may install, provided you are sufficiently creative with your package names. For more information on packages, consult Section 6.4 of the Python Tutorial online.

As you can see, creating modules and packages is rather straightforward. After creating a module you can import it any time you need to use it.

IMPORTING MODULES

Now that you know how to create modules, we can talk a little more about what actually happens during the importation process. As you have seen up to this point, the basic mechanism for loading a module is the **import** keyword. As you start working on your own modules, however, it is important to have a clearer understanding of exactly what happens when you import a module. In this section we briefly discuss the mechanics of the **import** keyword, as well as some special syntax that allows you finer control over your modules' namespaces.

import versus reload()

As we discussed earlier, the first time you import a module, a corresponding .pyc file will be generated. This .pyc file consists of bytecode that the interpreter actually reads at execution time. One important consequence of this implementation is that all subsequent importations of the module during a single session—from __main__ or any other module—will reference the compiled bytecode if it exists.

While this ensures that subsequent importations of the module into other namespaces are much faster than they might otherwise be, it also means that executable statements in the module are not repeated, and that any changes you make to the .py source file are not ready for use.

To handle these situations, you must use the **reload()** function. This function will recompile the bytecode for the module you pass as an argument (though it will not automatically recompile any modules that your module imports). Consequently, using **reload()** will not only repeat any executable

statements in the module, but will also update any attributes in the module to reflect any changes you may have made to its source code.

Note also that calling **reload()** in one namespace will not affect existing instances of the module in other namespaces. Because of how Python manages memory, other names will still point to old data (the module before it was reloaded). Moreover, reloading a package will not automatically reload all of its subpackages. Consult the description of **reload()** in Section 2 of Python Standard Library for more information.

It is important that MEL users contrast Python's **reload()** function and the MEL `source` directive. While **reload()** will both recompile and reexecute the specified module, the MEL `source` directive will only reexecute the contents of a supplied MEL script, but will not recompile it.[1]

The as Keyword

Sometimes a module is embedded deep within a package, or it has an inconveniently long (albeit descriptive) name. In addition to the package we created earlier, you have already seen one example of a module embedded within a package many times: maya.cmds. Unfortunately, prefixing every command with maya.cmds can quickly become tedious. The **as** keyword allows you to assign a custom name to the module when it is added to the global symbol table during importation. As you can see in the following code, this pattern can help you substantially reduce the amount of typing (and reading) you have to do, while still retaining the benefits of encapsulation.

```
import maya.cmds as cmds;
cmds.polySphere();
```

The from Keyword

Python also provides special syntax to import specific attributes from a module into the global symbol table. For example, the math module contains some useful trigonometric functions, such as **acos()**.

```
print(math.degrees(math.acos(0.5)));
```

If the module in which you are working does not have any conflicts in its namespace, you may find it helpful to just import these attributes directly into the global symbol table to access them without a prefix. Python allows

[1] It is possible to use the `eval` MEL command in conjunction with the `source` directive to trigger recompilation of a MEL script. Consult the MEL Command Reference document for more information.

you to accomplish this task using the **from** keyword. This pattern allows you to specify individual, comma-delimited attributes to import.

```
from math import degrees, acos;
print(degrees(acos(0.5)));
```

Python also allows you to import all attributes from a module into the current namespace using the asterisk character (*****).

```
from math import *;
print(degrees(asin(0.5)));
```

Remember to be cautious when you use such patterns! Python's system of namespaces is offered as a convenience and a safeguard, so do not dismiss it as a menace. Namespaces can protect you from any unforeseen conflicts, especially when you are working on large projects with other developers. For example, you may be importing all of a module's attributes into your global symbol table, and someone working on the other module may later add an attribute that conflicts with one in your own module. In many cases, if verbosity is causing problems, it is often safest to use the **as** keyword and assign a short name to the module you are importing.

Attribute Naming Conventions and the from Keyword

Before moving on to create our own modules, it is worth pointing out the naming convention of some of the attributes we have seen so far. Recall that we mentioned that attributes in Python are not private, as the concept exists in other languages such as C++. We can still mutate modules while inside some other module.

In Python, many attributes are prefixed with underscores, indicating that they are "internal" symbols. In addition to essentially being a suggestion that you treat them as though they were private (by rendering them inconvenient to type, if nothing else), this convention also ensures that the attributes are not imported into the global symbol table when using the syntax to import all of a module's attributes with the **from** keyword and the asterisk. You certainly wouldn't want to overwrite the **__name__** attribute of the __main__ module with the name of another module you were importing!

PYTHON PATH

In MEL, you are able to call the `source` directive and pass a path to a script located anywhere on your computer to load it, even if it is not in your MEL script path. Importing in Python has no analogous option. When you attempt to import a module, Python searches all of the directories that have been specified in your Python path. In Python, all modules must be located in one of

these directories, or inside of a package in one of these directories, and you have a variety of options available for appending directories to your Python path. In this section we will look more closely at different ways you can manipulate your Python search path.

sys.path

One handy mechanism for working with your Python path is the **path** attribute in the sys module. This attribute contains a list of all directories that Python will search, and you can interactively work with it like any other list, including appending to it.

```
import sys;
for p in sys.path: print(p);
```

If you import the sys module and print the elements in **path**, you may see something like the following example (in Windows).

```
C:\Program Files\Autodesk\Maya2012\bin
C:\Program Files\Autodesk\Maya2012\bin\python26.zip
C:\Program Files\Autodesk\Maya2012\Python\DLLs
C:\Program Files\Autodesk\Maya2012\Python\lib
C:\Program Files\Autodesk\Maya2012\Python\lib\plat-win
C:\Program Files\Autodesk\Maya2012\Python\lib\lib-tk
C:\Program Files\Autodesk\Maya2012\bin
C:\Program Files\Autodesk\Maya2012\Python
C:\Program Files\Autodesk\Maya2012\Python\lib\site-packages
C:\Program Files\Autodesk\Maya2012\bin\python26.zip\lib-tk
C:/Users/Adam/Documents/maya/2012/prefs/scripts
C:/Users/Adam/Documents/maya/2012/scripts
C:/Users/Adam/Documents/maya/scripts
```

The first several of these directories are built-in paths for Maya's Python interpreter, and they are all parallel to Maya's Python install location. Depending on your platform, these are:

- **Windows:** C:\Program Files\Autodesk\Maya<version>\…
- **OS X:** /Applications/Autodesk/maya<version>/Maya.app/Frameworks/…/lib/…
- **Linux:** /usr/autodesk/maya/lib/…

If you run a separate Python interpreter on your machine, you would see similar directories if you were to print this same example. These directories are all made available as soon as your Python session has begun. The final directories in the list are specific to Maya. Depending on your platform, they will be something like the following:

- **Windows:** C:\Users\<username>\Documents\maya\…

- **OS X:** /Users/<username>/Library/Preferences/Autodesk/maya/…
- **Linux:** /home/<username>/maya/…

These final directories are appended to the Python path when Maya is initialized.[2] It is important to know how the Python path is altered during the launch process in some cases. For example, if you launch Maya's interpreter, mayapy, on its own to use Maya without a GUI, this second group of paths is only accessible after you have imported the maya.standalone module and called its **initialize()** function. This order of operations is also important if you want to deploy a sitecustomize module, which we will discuss shortly. For now, we turn our attention to the most basic way to add directories to your search path.

Temporarily Adding a Path

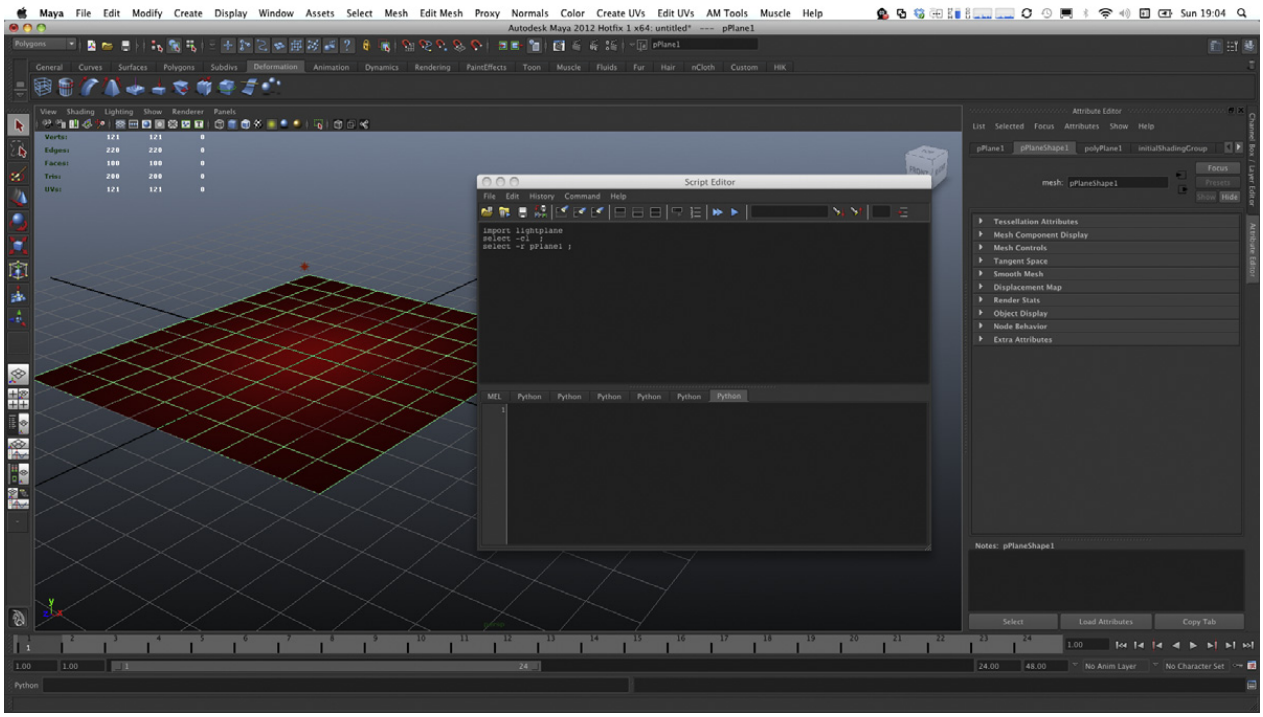
1. As we mentioned earlier, because the **path** attribute is simply a list, you can easily append to it just as you can any other list. You can take advantage of this functionality to temporarily add a directory to your Python path during a Maya session. We'll show a quick example to add a module to your desktop and then add your desktop to your Python search path to import the module. Download the lightplane.py module from the companion web site and save it to your desktop. The module creates a red light and a plane in your scene.
2. Now append your desktop to the **sys.path** attribute by executing the following lines in the Script Editor.

```
import os, sys;
home = os.getenv('HOME');
user = os.getenv('USER');
sys.path.append(
    os.path.join(
        home[:home.find(user)],
        user, 'Desktop'
    )
);
```

In this case, we were able to take advantage of a useful built-in module, os, to help build the path. The first two calls return the value of environment variables with the specified keys, the first providing a path to the user's home folder and the second returning the name of the user.

The os module has a path extension module with a **join()** function, which concatenates multiple strings using whatever directory delimiter your operating system uses (forward slash on OS X and Linux, backslash on

[2]Note that directories in your **sys.path** attribute are supplied as absolute paths, and do not include the tilde (~) character for OS X and Linux as a shortcut for the user's home directory.



■ **FIGURE 4.4** The results of the lightplane module.

Windows).[3] Unfortunately, in this example we also have to do some slicing on the home directory, as it points to My Documents in Windows rather than to the user's folder. If you like, you can print the entries in the **sys.path** attribute to see the result for your platform at the end of the output.

3. Now that you have added your desktop to your path, you can import the lightplane module, after which you should see something like Figure 4.4 if you enable lighting (press **5** on the keyboard to switch to shaded view, and then **7** on the keyboard to enable lighting).

```
import lightplane;
```

If you simply append a location to the **sys.path** attribute, it will only be in effect until Maya closes. While this may be handy if you need to add a location from within some other module, or if you need to test something that you haven't yet buried away in your Python path, you will often want some directories to be consistently added to your Python path.

[3]Note that newer versions of Windows use both delimiters fairly interchangeably. Python can handle them both just fine, but it's always safest to use the os.path module to build paths.

While you have a variety of options for automatically configuring your Python search path across all your sessions, we cover three of the most common options here: creating a userSetup script, creating a sitecustomize module, and configuring an environment variable. In some cases, you may find that a combination of these approaches meets your particular needs based on how your studio deploys tools.

userSetup Scripts

A useful feature of Maya is the ability to create a userSetup script. A userSetup script simply needs to be in a proper location on your computer, and Maya will execute any instructions in it when it starts. Depending on your operating system, you can add such a script to the specified default directory:

- **Windows:** C:\Documents and Settings\user\My Documents\maya\<version>\scripts\
- **OS X:** ~/Library/Preferences/Autodesk/maya/<version>/scripts/
- **Linux:** ~/maya/<version>/scripts/

Note that you may only have one userSetup script and it must be either a Python or a MEL script—you cannot have both a userSetup.mel and a userSetup.py script. A MEL script will take precedence if both exist.

In the following brief example, you will alter your userSetup script to automatically append your desktop to the Python search path and import the lightplane module that you created in the previous example (which should still be on your desktop).

1. If you do not have a userSetup script, create either a userSetup.mel or a userSetup.py script in the appropriate directory for your operating system. Copy the contents of the appropriate following example into your script.

userSetup.py

```
import os, sys;
home = os.getenv('HOME');
user = os.getenv('USER');
sys.path.append(
    os.path.join(
        home[:home.find(user)],
        user, 'Desktop'
    )
);
import lightplane;
```

userSetup.mel

```
python("import os, sys");
python("home = os.getenv('HOME')");
```

```
python("user = os.getenv('USER')");
python("sys.path.append(os.path.join(home[:home.find
(user)], user, 'Desktop'))");
python("import lightplane");
```

2. Close Maya and reopen it to start a new session.

Contrary to what you may have expected, the scene does not in fact contain the plane and red light! What happened? In essence, Maya imported your module before the Maya scene was prepared. Nonetheless, you can confirm that your userSetup script was properly executed by printing your global symbol table.

```
for k in globals().keys(): print(k);
```

You should see `os`, `sys`, and `lightplane` among the names.

```
__builtins__
__package__
sys
user
home
lightplane
__name__
os
__doc__
```

It is interesting to note that a userSetup script is not fully encapsulated like an ordinary module. All of its statements are executed in the __main__ module using the built-in **execfile()** function, so any modules you import from userSetup are automatically available in __main__ when Maya is initialized, as are any attributes you define.[4]

3. Return to your userSetup script and replace the lines you entered in step 1 with the following (we assume MEL users can translate easily enough).

```
import maya.utils, os, sys;
home = os.getenv('HOME');
user = os.getenv('USER');
sys.path.append(
    os.path.join(
        home[:home.find(user)],
        user, 'Desktop'
    )
);
maya.utils.executeDeferred('import lightplane');
```

[4]If you want to know more, you can investigate the modules that are part of Maya's installation. The userSetup script is executed in site-packages/maya/app/startup/basic.py.

As you can see, the only real change we made was to wrap the importation of the lightplane module in a call to the **executeDeferred**() function in the maya.utils module. This function allows you to queue up operations to be executed as soon as Maya is idle, which will be after the scene is initialized in this case.

4. Close Maya and reopen it. You should now see the red light and polygon plane in your scene.

While the userSetup script is a convenient way to automatically perform some operations, many Maya users are also accustomed to using it for their own purposes. You could wind up with complications if you require your artists to configure their path in their userSetup scripts, as opposed to simply importing your studio's tool modules. As such, a userSetup script is not always the best way to set up your path. Fortunately, it is also not the only option you have available.

sitecustomize Module

A built-in feature in Python is support for a sitecustomize module. A sitecustomize module is always executed when your Python session begins. Similar to a userSetup script, this module allows you to import modules, append to **sys.path**, and execute statements. *However, unlike a userSetup script, a sitecustomize module encapsulates its attributes like any other module, and so any modules you import in it are not automatically in the global symbol table for __main__*. We will see this issue in the next example.

The principle advantage of using a sitecustomize module is that it offers you an opportunity to set up the Python path without interfering with a userSetup script. Consequently, you could deploy a studiowide sitecustomize module to configure users' paths to point to network directories and so on. At the same time, your artists would still be able to set up their own individual userSetup scripts, as either MEL or Python depending on their needs and comfort levels. Artists would only need to import one package for your tools, and would be free to load other tools they may have downloaded from the Internet. You can offer them the flexibility to configure their environments individually, while also reducing the likelihood that they will inadvertently affect their search paths.

Recall earlier when we printed the directories in **sys.path** that there were essentially two groups of folders: those belonging to Maya's Python interpreter, and those that are added during Maya's initialization process. *Your sitecustomize module is imported before Maya's directories are added to the search path*. Consequently, convention dictates that you should put your sitecustomize module in your site-packages directory, which is in your

Python path by default. (In practice, you are free to include it in any directory that is part of your search path prior to Maya's initialization.) In this brief example, you will simply recreate the path manipulation functionality of the userSetup script that you created in the previous example.

1. Navigate back to the folder where you put your userSetup script, and remove the lines from it that you added in the previous example, or delete it altogether.

sitecustomize.py

```
import os, sys;
home = os.getenv('HOME');
user = os.getenv('USER');
sys.path.append(
    os.path.join(
        home[:home.find(user)],
        user, 'Desktop'
    )
);
```

2. Create a new module called sitecustomize.py and copy the contents of our example printed here into it. Save it in your site-packages directory in Maya's installation location. Note that to reach this location in OS X, you must press **Cmd-Shift-G** from a Save dialog to type in the path (you can also right-click on Maya.app and select Show Contents to drag a file into this location in Finder). Windows users may have to supply administrative privileges to write to this folder. The location of this folder varies based on your operating system:
   - **Windows:** C:\Program Files\Autodesk\Maya<version>\Python\lib\site-packages
   - **OS X:** Applications/Autodesk/maya<version>/Maya.app/Contents/Frameworks/Python.framework/Versions/Current/lib/python<version>/site-packages
   - **Linux:** /usr/autodesk/maya/lib/python<version>/site-packages
3. Close and reopen Maya to start a new session.
4. If you print the contents of your **sys.path** attribute, you will see that your desktop has been added to the search path immediately after the site-packages folder.

```
import sys;
for p in sys.path: print(p);
```

For example, a Windows user may see the following output.

```
C:\Program Files\Autodesk\Maya2012\bin
C:\Program Files\Autodesk\Maya2012\bin\python26.zip
```

```
C:\Program Files\Autodesk\Maya2012\Python\DLLs
C:\Program Files\Autodesk\Maya2012\Python\lib
C:\Program Files\Autodesk\Maya2012\Python\lib\plat-win
C:\Program Files\Autodesk\Maya2012\Python\lib\lib-tk
C:\Program Files\Autodesk\Maya2012\bin
C:\Program Files\Autodesk\Maya2012\Python
C:\Program Files\Autodesk\Maya2012\Python\lib\site-packages
C:/Users/Adam\Desktop
C:\Program Files\Autodesk\Maya2012\bin\python26.zip\lib-tk
C:/Users/Adam/Documents/maya/2012/prefs/scripts
C:/Users/Adam/Documents/maya/2012/scripts
C:/Users/Adam/Documents/maya/scripts
```

At this point, you could import the lightplane module if you like. However, there are a couple of important points worth discussing here. First, notice that the path to your desktop immediately follows the site-packages folder, indicating that the adjustments made in sitecustomize.py become immediately available—before Maya has initialized. Consequently, you cannot issue Maya commands from a sitecustomize module. Although you would be able to import the cmds module without errors (since it is accessible from the site-packages directory), it is effectively a dummy at this point in the initialization process, containing only placeholders for Maya commands. The maya.utils module suffers from the same limitation in this case.

Second, unlike the userSetup script, sitecustomize operates as an ordinary module, and therefore encapsulates its data. Consequently, if you were to print the global symbol table for the __main__ module at this point, you would not see any of the modules you imported or other variables you defined from within sitecustomize.py, such as `os` or the `home` or `user` variables.

While using a sitecustomize module is handy, you may find it inconvenient to write to a folder in Maya's install location, especially if you need to update the module occasionally. Another tool at your disposal for configuring your Python path, which can work in conjunction with a sitecustomize module and/or a userSetup script, is to set up an environment variable.

Setting Up a PYTHONPATH Environment Variable

Setting up an environment variable allows you to specify directories in your Python path that will be available before Maya has initialized, thus eliminating the need to append to **sys.path**. You can set up an environment variable specifically for Maya, or you can set up one that is systemwide. Either way, because directories you add via an environment variable are part of the search path before Maya has initialized, you can also put your sitecustomize module in a folder that you have specified using an environment variable, if you so choose.

Maya-Specific Environment Variables with Maya.env

The easiest way to configure an environment variable is to set up one that is exclusive to Maya. Maya has a file, Maya.env, available in a writable location, which allows you to configure a host of environment variables. You can add an environment variable, PYTHONPATH, to this file, and set its value to a collection of directories you want to include in the search path.

1. Delete the sitecustomize.py file that you created in the previous example if you still have it.
2. Open the Maya.env file in a text editor. Depending on your operating system, you can find this file in one of the following locations:
   - **Windows:** C:\Documents and Settings\user\My Documents\maya\<version>
   - **OS X:** ~/Library/Preferences/Autodesk/maya/<version>
   - **Linux:** ~/maya/<version>
3. By default, your Maya.env file will be blank. This file allows you to specify environment variables and their values. Set a value for the PYTHONPATH variable to point to your desktop, where you saved the lightplane module. The syntax is similar for all platforms; just ensure you set the value appropriately based on your operating system. For example, if you are a Windows user, you may enter the following line.

   ```
   PYTHONPATH = C:\Users\Adam\Desktop
   ```

   If you already have a value set for PYTHONPATH, or if you want to add more directories, simply add your operating system's delimiter between them (Windows uses a semicolon, Linux and OS X use a colon). For example, a Windows user could enter the following.

   ```
   PYTHONPATH = C:\Users\Adam\Desktop;C:\mypythonfiles
   ```

4. Save your changes to the Maya.env file.
5. Close and reopen Maya to start a new session.
6. Open the Script Editor and print the **sys.path** attribute.

   ```
   import sys;
   for p in sys.path: print(p);
   ```

   As you can see, your desktop is now part of the Python search path—even before many of the default directories.

   ```
   C:\Program Files\Autodesk\Maya2012\bin
   C:\Users\Adam\Desktop
   C:\Program Files\Autodesk\Maya2012\bin\python26.zip
   C:\Program Files\Autodesk\Maya2012\Python\DLLs
   C:\Program Files\Autodesk\Maya2012\Python\lib
   ```

```
C:\Program Files\Autodesk\Maya2012\Python\lib\plat-win
C:\Program Files\Autodesk\Maya2012\Python\lib\lib-tk
C:\Program Files\Autodesk\Maya2012\bin
C:\Program Files\Autodesk\Maya2012\Python
C:\Program Files\Autodesk\Maya2012\Python\lib\
site-packages
C:\Program Files\Autodesk\Maya2012\bin\python26.zip\
lib-tk
C:/Users/Adam/Documents/maya/2012/prefs/scripts
C:/Users/Adam/Documents/maya/2012/scripts
C:/Users/Adam/Documents/maya/scripts
```

At this point, you could again import the lightplane module without having to alter the **sys.path** attribute.

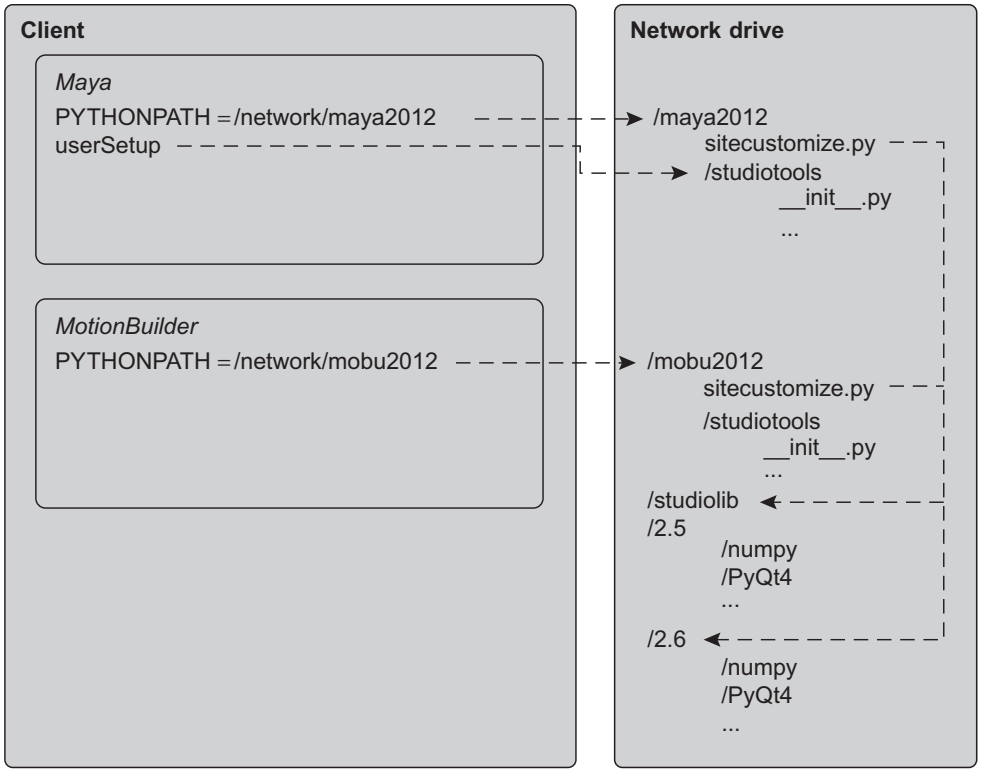
Modifying the Maya.env file is one of the easiest and most reliable ways to configure your Python search path with minimal effects on other users. In a majority of cases, using some combination of the techniques we have discussed thus far is perfectly sufficient. However, you can also create a systemwide PYTHONPATH variable.

Systemwide Environment Variables and Advanced Setups

Your operating system allows you to define many custom environment variables that are shared across all applications. Because setting a systemwide environment variable will override your settings in Maya.env, and because it will apply to all applications that make use of it, it can be dangerous to use one, but also quite powerful if you know what you are doing. Systemwide environment variables simply offer you one more option for deploying common code across a range of applications.

For example, in addition to your Maya tools, you may have some shared libraries (e.g., numpy, PyQt4, and some Python utilities) that you use across a range of applications (e.g., Maya, MotionBuilder, and so on). To avoid deploying these modules to individual users, you may put them on a network drive, along with your tools for each specific application.

One option for solving this problem is to modify application-specific environment variables for each interpreter to point to individual sitecustomize modules in network directories (one directory/sitecustomize for each version of each application). In this scenario, each application has its own PYTHONPATH variable pointing to a network directory where you can easily deploy and update a sitecustomize module that appends new directories to the path (Figure 4.5). Each sitecustomize module could append further network directories relevant to the application's Python version and so on.

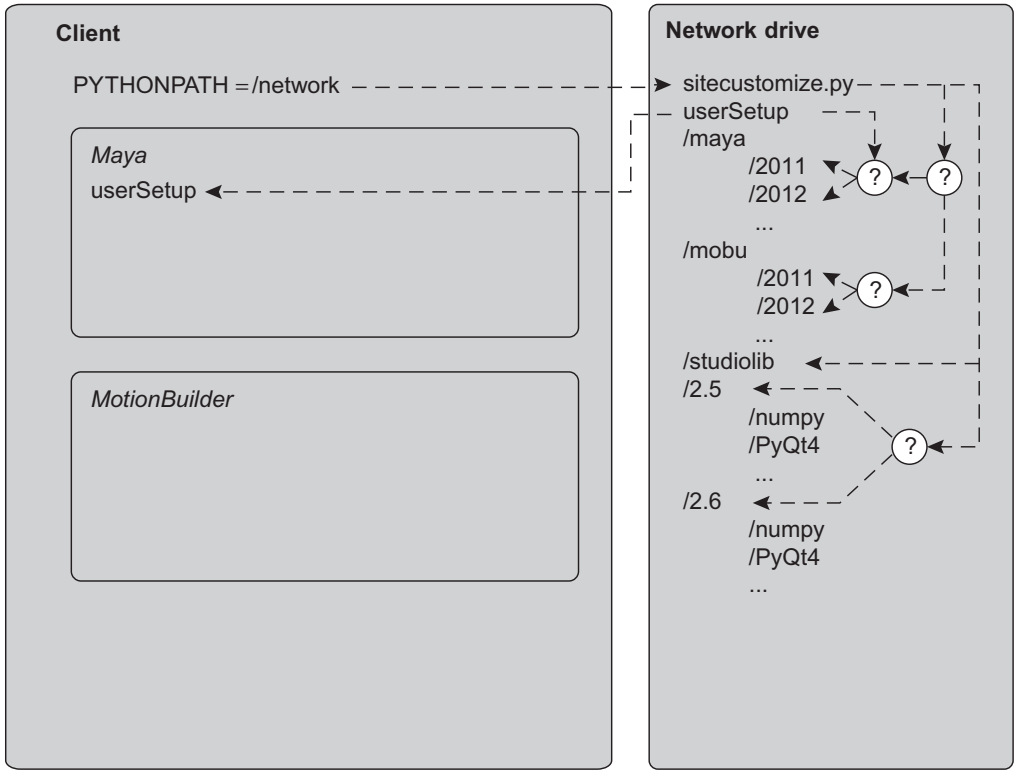


■ **FIGURE 4.5** One way to deploy systemwide modules with separate sitecustomize modules.

However, you may find it troublesome to add new software to your pipeline, or to upgrade existing software. For example, if you upgrade to a new version of Maya with a new version of Python, you would need to make sure you were pointing to a new version of PyQt and so on.

You could also set up a sitecustomize module on a network folder that is specified via a systemwide environment variable (Figure 4.6). Using a setup with systemwide environment variables would also enable you to set values for MAYA_SCRIPT_PATH, which is used to locate MEL scripts. Such a configuration would allow you to take advantage of the search path's construction order to use global userSetup scripts for your artists. Using a global userSetup script, you could automatically import your tools and then search for an individual user's userSetup script to execute it as well, which would allow your artists complete freedom over their own userSetup scripts without you having to worry that they import your tools.

In this setup, your global sitecustomize module could use attributes in the sys module to find out what application is importing it (**sys.executable**)



■ **FIGURE 4.6** One way to deploy systemwide modules with a systemwide environment variable.

and what version of Python it is running (**sys.version**). It could then use this information to append directories to the search path as needed. In this setup, any time you add a new piece of software to your pipeline, it can access your shared libraries right away. Keeping the sitecustomize module in one place where you can write to it makes it easy to update your whole team's toolset immediately. Just don't take that kind of power lightly! (If you are feeling really creative—or cavalier—you can also import __builtin__ in your sitecustomize module and add attributes to it for modules you know you will consistently want available right away, such as math, sys, os, or re.)

You can find more information about systemwide environment variables on the Internet, but we offer a few quick pointers here to get you started if you are interested.

Windows

To create a systemwide environment variable, open your System Properties dialog from the Control Panel. In the Advanced tab, there is an Environment Variables button at the bottom of the dialog. Clicking this button opens

a dialog that lets you create new variables that apply to the current user as well as variables that apply to the system. Simply create a variable called PYTHONPATH in the User section and supply it with semicolon-delimited directories just as you did in Maya.env. When you press OK, the new variable is ready to go and in place for all Python interpreters you run.

OS X

Apple's official technical documentation tells users to add environment variables to an environment.plist file and put it in the ~/.MacOSX directory. The problem with setting up an environment variable in this way is that it will only apply to applications with particular parent processes. For example, if you were to configure PYTHONPATH in this way and launch Maya from Spotlight rather than Finder, it would not be in effect.

To create an environment variable in OS X that will apply to applications with any parent process, you need to modify the configuration file for the launchd daemon. If you do not have one, you can create one at /etc/launchd.conf. Inside this text file, you can add a line like the following example with colon-delimited directories to specify your Python path.

```
setenv PYTHONPATH /users/Shared/Python:/users/Shared/
MorePythonStuff
```

USING A PYTHON IDE

As you start creating more modules and complex packages, the ability to work efficiently becomes more critical. While text editors and the Maya Script Editor are fine tools for quickly experimenting with code, they're not optimal environments for dealing with complex projects, version control systems, and so on. A better option is to use a Python IDE.

An IDE, or integrated development environment, is a tool for programming in large projects efficiently. Some major Python IDEs include features such as text editors with syntax highlighting and automatic code completion, debuggers, project navigation views, in-editor version control support for tools like SVN and Git, integrated interpreters, and more. In short, if you plan on writing more than one short Python script per year, you probably stand to gain a great deal by using a Python IDE.

Downloading an IDE

While you have many options available for coding in Python, we will only discuss two popular options here: Wing IDE and Eclipse. We only cover basics to save space, but you can consult the companion web site for links

and more information on how to set up these tools. Our main goal here is simply to help you find both of the tools and briefly examine some of their advantages and disadvantages. Knowing the strengths and weaknesses of each tool can help you find which is most comfortable for you, or even use them effectively in concert with one another!

Wing IDE

Wing IDE is a commercial product developed by Wingware, and is available at *http://wingware.com/*. As a commercial product, Wingware benefits from being a fairly straightforward tool to install, configure, and use. Moreover, Wingware offers a personal version, as well as a student version of Wing IDE, each of which is available at a lower price point than the professional version, yet which omit some features. Consequently, many users often find that Wing IDE is the easiest entry point when they begin working.

Wingware offers a handy online guide to setting up Wing IDE for use with Maya, which includes information on configuring its debugger, code completion, and sending commands to Maya (Wing uses its own interpreter by default).

Eclipse

Eclipse is an open-source product, and is available at *http://www.eclipse.org/downloads/*. Like many open-source products, Eclipse may seem daunting at first to those who are generally only familiar with commercial software. For starters, going to the download page offers a range of options, as the community has customized many configurations for common uses. (Note that Eclipse originated primarily as a Java development platform.) In our case, we are interested in Eclipse Classic. Once you have installed Eclipse, you need to download and install the Pydev plug-in, which can be done directly from the Eclipse GUI.

While Eclipse can perhaps be more complicated to configure, it does have some benefits. The most obvious benefit is of course that it is free. Moreover, Eclipse allows you to fully configure a variety of interpreters and select which interpreter each project is using. Furthermore, because the Eclipse community has developed a range of free plug-ins, you can easily download and install additional tools from within the application.

Basic IDE Configuration

Once you have downloaded and installed one or more IDEs to try out, you will need to go through a few steps to configure your environment. While we leave the specific details for the companion web site, we have selected a

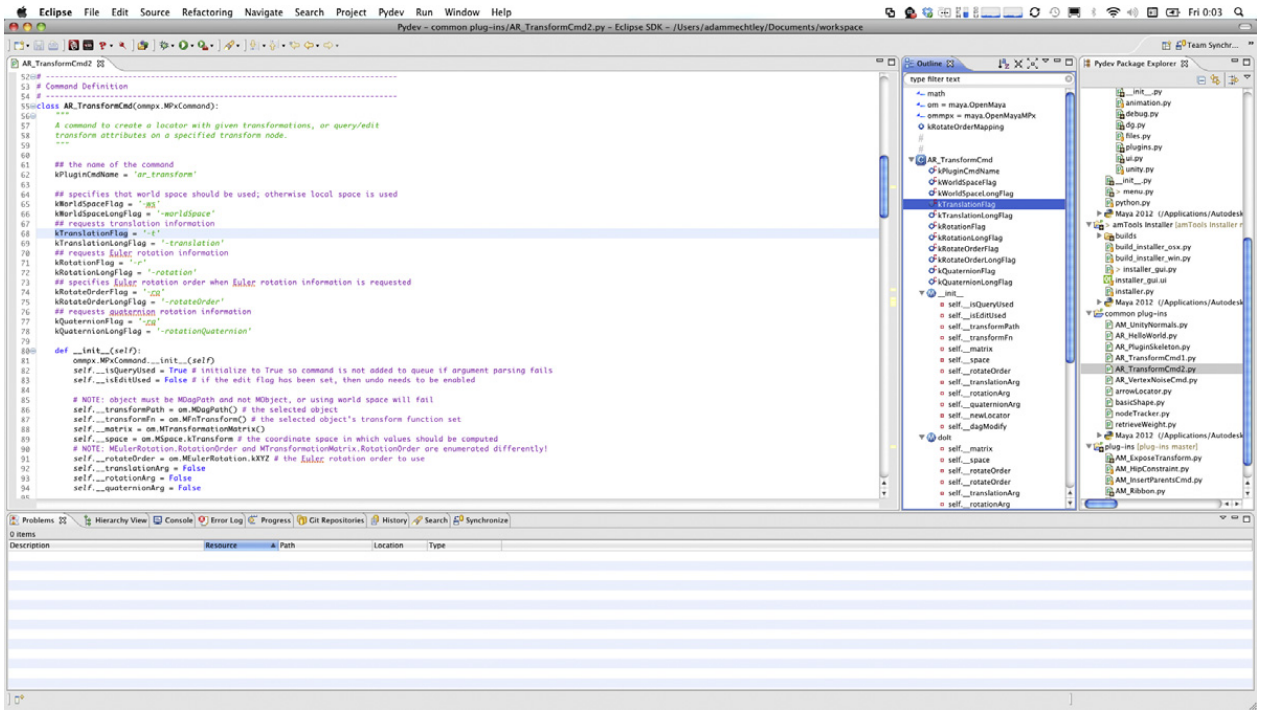
few high-level points to discuss here so you have a better idea of what is going on under the hood.

Views

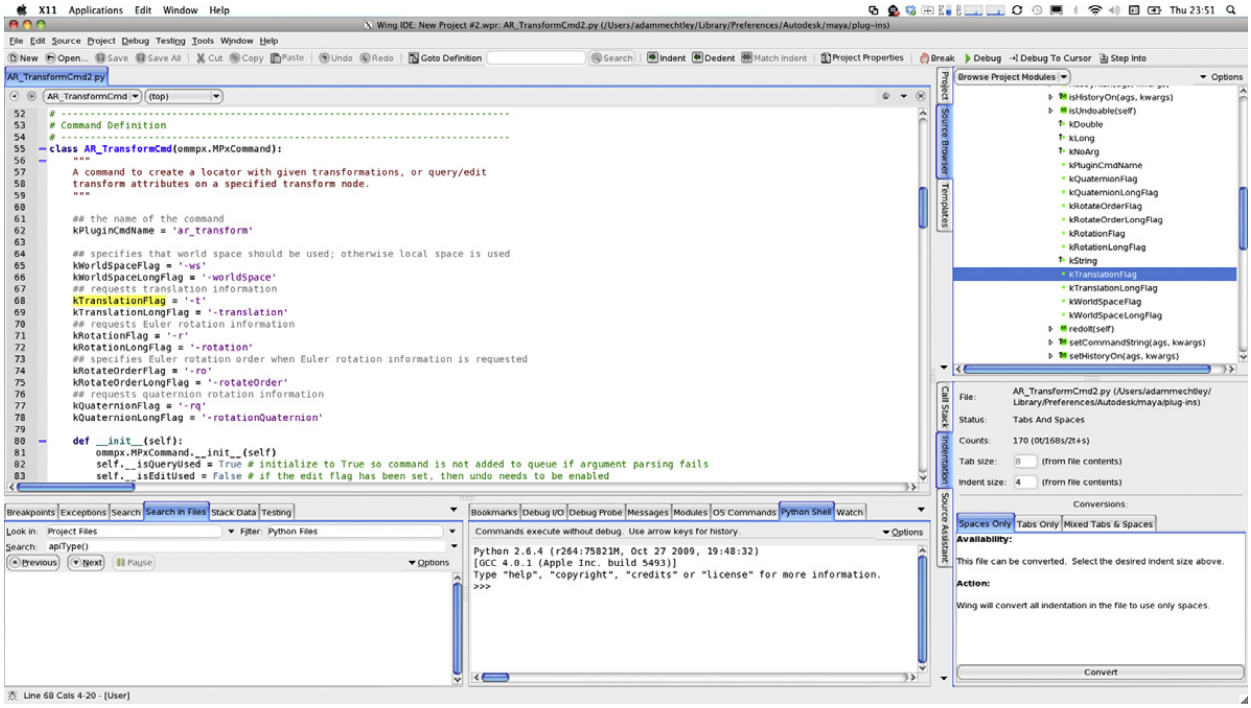
Whatever IDE you are using will be made up of multiple views, which can usually be customized into a layout that works for you. Figures 4.7 and 4.8 show some of the basic panels in Eclipse and Wing IDE, respectively. The largest panel is the text editor, where you actually input code (center-left in both of the screenshots).

Each IDE also contains project views, which display all of the modules and packages in your project. In our screenshots, Wing IDE's source browser is occluding the project view (right), but you can see the tab to expose it. Eclipse's project browser, on the other hand (far right), displays several projects, each expanded to show their contents. Moreover, each project also displays what interpreter it is using.

Another helpful feature in many IDEs is a source browser or module outliner (right in both). These views show you a collapsible outline of the current module, including its classes and other attributes. In both Wing IDE



■ **FIGURE 4.7** Eclipse.



■ **FIGURE 4.8** Wing IDE.

and Eclipse, selecting an item from the outliner allows you to quickly jump to its definition, as well as see its scope. Moreover, Eclipse has a handy built-in feature to highlight all occurrences of the selected object, which not only highlights it in the text editor view but also highlights markers on the text editor's scrollbar to see where other occurrences are.

Each editor has a number of other views (which we will not detail here), including panels for debugging, interactively using an interpreter, reviewing project errors, console output, version control history, and much more.

Selecting an Interpreter

Most Python IDEs integrate an interpreter that you can use for executing code, debugging, and so forth. Wing IDE is bundled with its own interpreter, but Eclipse allows you to configure as many as you like, using interpreters already existing on your machine. Because Wing IDE ships with its own interpreter, you cannot directly use Maya's modules in its interactive terminal to test them.

An advantage to configuring multiple interpreters is that you may deploy to different versions of Python across the different products you are

supporting, whether they are different versions of Maya or other applications like MotionBuilder. As such, you can ensure that your projects are properly implementing supported language features for the end user's environment, for example. Moreover, you can individually configure the search path for each interpreter you configure. In Eclipse, you can configure interpreters from the **Preferences** menu, where you can select **Interpreter – Python** in the Pydev dropdown menu. You can consult the companion web site for more information on configuring an interpreter for Maya in Eclipse.

Automatic Code Completion

Most advanced Python development environments will support some mechanism for autocompletion of code. The basic idea is that the IDE is aware of the contents of modules you import, and so it can begin to offer completion suggestions while you type. As you can imagine, this feature can be a huge time saver and can reduce typos.

While most IDEs can offer interactive completion by analyzing the contents of the module in which you are working, as well as other modules in your project, this mode of autocompletion may be sluggish. As such, IDEs such as Eclipse and Wing IDE allow you to use what are called stubs to generate autocompletion results for modules. In short, the stubs are special, concise files with either a .py or .pi extension, which contain dummies for some actual module, allowing them to be parsed for autocompletion more quickly. Fortunately, there are handy tools for automatically generating these stubs from Maya's modules. You can find more information about this process on the companion web site.

Projects

The key mode of organization in a Python IDE is to create projects. Projects are typically small description files that contain information about a collection of modules (e.g., their location on disk), the interpreter to be used, version control settings, and so on. Consequently, project files do not usually store any of your actual data, but rather allow you to easily locate and organize your files. Many IDEs also offer special tools for performing projectwide searches, replacements, and so on.

Connecting to Maya

Both Wing IDE and Eclipse offer functionality for advanced users to connect the IDE directly to Maya. The basic process is that you can call the `commandPort` command in your userSetup script to automatically listen

for input from a port on which the IDE will send commands to Maya. Using this feature enables you to execute a whole module (or individual lines of code that you select) and send them to Maya's __main__ module, just like using the Script Editor. You can refer to the companion web site for more information on tools for connecting to Maya.

Debugging

The final important feature we want to note is that many IDEs, such as both Wing IDE and Eclipse, offer tools for debugging modules. Using an integrated debugger allows you to set break points in your module before you execute it, so you can interactively monitor the values of different variables during execution. You can refer to the companion web site for links to information on the debuggers for Wing IDE and Eclipse.

CONCLUDING REMARKS

You now have a solid grasp of modules and packages, as well as how to optimize your development and deployment practices. Although we introduced some of Python's built-in modules in this chapter, such as math, os, and sys, there are many more that you will find useful when working in Maya. We encourage you to consult Python's online documentation and familiarize yourself with some of the most common modules, as they have often already solved problems you may otherwise toil over! Likewise, there are many useful modules in the Python community at large that you can use in Maya.

Although the power of modules should be clear, particularly compared to traditional MEL scripts, they are still only the tip of the iceberg for what you can accomplish using Python in Maya. Now that you are better acquainted with concepts such as scope and encapsulation, you are ready to start working with one of Python's greatest advantages over MEL: the ability to do object-oriented programming.