Modules and Packages for Code Reuse

Up until this chapter, we have been looking at code at the level of the interactive console and simple scripts. This works well for small examples, but when your program gets larger, it becomes necessary to break programs up into smaller units. In Jython, the basic building block for these units in larger programs is the module.

Imports for Reuse

Breaking code up into modules helps to organize large code bases. Modules can be used to logically separate code that belongs together, making programs easier to understand. Modules are helpful for creating libraries that can be imported and used in different applications that share some functionality. Jython's standard library comes with a large number of modules that can be used in your programs right away.

Import Basics

The following is a very simple program that we can use to discuss imports.

breakfast.py

```
import search.scanner as scanner
import sys
class Spam(object):

    def order(self, number):
        print "spam " * number

    def order_eggs():
        print " and eggs!"

    s = Spam()
    s.order(3)
    order_eggs()
```

We'll start with a couple of definitions. A **namespace** is a logical grouping of unique identifiers. In other words, a namespace is that set of names that can be accessed from a given bit of code in your program. For example, if you open up a Jython prompt and type dir(), the names in the interpreter's namespace will be displayed.

```
>>> dir()

['__doc__', '__name__']
```

The interpreter namespace contains __doc__ and __name__. The __doc__ property contains the top level docstring, which is empty in this case. We'll get to the __name__ property in a moment. First we need to talk about Jython **modules**. A **module** in Jython is a file containing

Python definitions and statements which in turn define a namespace. The module name is the same as the file name with the suffix .py removed, so in our current example the Python file 'breakfast.py' defines the module 'breakfast.'

Now we can talk about the __name__ property. When a module is run directly, as in jython breakfast.py, __name__ will contain '__main__'. If a module is imported, __name__ will contain the name of the module, so 'import breakfast' results in the breakfast module containing a __name__ of 'breakfast'. Again from a basic Jython prompt:

```
>>> dir()
['__doc__', '__name__']
>>> __name__
'__main__'
```

Let's see what happens when we import breakfast:

```
>>> import breakfast
spam spam spam
and eggs!
>>> dir()
['__doc__', '__name__', 'breakfast']
>>> import breakfast
>>>
```

Checking the dir() after the import shows that breakfast has been added to the top level namespace. Notice that the act of importing actually executed the code in breakfast.py. This is the expected behavior in Jython. When a module is imported, the statements in that module are actually executed. This includes class and function definitions. It is important to note that this only happens the first time you import a module. Note the last statement where we issue 'import breakfast' again, resulting in no output. Most of the time, we wouldn't want a module to execute print statements when imported. To avoid this, but allow the code to execute when it is called directly, we typically check the __name__ property. If the __name__ property is '__main__', we know that the module was called directly instead of being imported from another module.

```
class Spam(object):

    def order(self, number):
        print "spam " * number

    def order_eggs():
        print " and eggs!"

if __name__ == '__main__':
s = Spam()
s.order(3)
order_eggs()
```

Now if we import breakfast (remember to close and reopen the interpreter so that the module is actually reimported!), we will not get the output:

```
>>> import breakfast
```

This is because in this case the __name__ property will contain 'breakfast,' the name of the module. If we call breakfast.py from the commandline like 'jython breakfast.py' we would then get the output again, because breakfast would be executing as __main__:

```
$ jython breakfast.py
spam spam spam
and eggs!
```

The Import Statement

In Java, the Import statement is strictly a compiler directive that must occur at the top of the source file. In Jython, the import statement is an expression that can occur anywhere in the source file, and can even be conditionally executed.

As an example, a common idiom is to attempt to import something that may not be there in a try block, and in the except block define the thing in some other way, or import it from a module that is known to be there.

```
>>> try:
...     from blah import foo
...     print "imported normally"
... except ImportError:
...     print "defining foo in except block"
...     def foo():
...         return "hello from backup foo"
...
defining foo in except block
>>> foo()
'hello from backup foo'
>>>
```

If a module named 'blah' had existed, the definition of foo would have been taken from there and we would have seen 'imported normally' printed out. Because no such module existed, foo was defined in the except block, 'defining foo in except block' was printed, and when we called foo, the 'hello from backup foo' string was returned.

An Example Program

Here is the layout of a contrived but simple program that we will use to describe some aspects of importing in Jython.

```
chapter8/
```

```
        greetings.py
        greet/
                __init__.py
                hello.py
                people.py
```

This example contains one package: greet, which is a package because it is a directory containing the special __init__.py file. Note that the directory chapter8 itself is not a package because it does not contain an __init__.py. There are three modules in the example program: greetings, greet.hello, and greet.people. The code for this program can be downloaded at http://kenai.com/projects/jythonbook/sources/jython-book/show/src/chapter8.

greetings.py

```
print "in greetings.py"
import greet.hello

g = greet.hello.Greeter()
g.hello_all()
```

greet/__init__.py

```
print "in greet/__init__.py"
```

greet/hello.py

```
print "in greet/hello.py"
import greet.people as people

class Greeter(object):
    def hello_all(self):
        for name in people.names:
            print "hello %s" % name
```

greet/people.py

```
print "in greet/people.py"

names = ["Josh", "Jim", "Victor", "Leo", "Frank"]
```

Trying Out the Example Code

If you run greetings.py in its own directory you get the following output:

```
$ jython greetings.py
in greetings.py
in greet/__init__.py
in greet/hello.py
in greet/people.py
hello Josh
hello Jim
hello Victor
hello Leo
hello Frank
```

There is a print statement at the top of each of the .py files to show the order of execution for the modules. When run, the module greetings is loaded, printing out 'in greetings.py.' Next it imports greet.hello:

```
import greet.hello
```

Because this is the first time that the greet package has been imported, the code in __init__.py is executed, printing 'in greet/__init__.py'. Then the greet.hello module is executed, printing out 'in greet/hello.py.' The greet.hello module then imports the greet.people module, printing out 'in greet/people.py.' Now all of the imports are done, and greetings.py can create a greet.hello.Greeter class and call its hello_all method.

Types of Import Statements

The import statement comes in a variety of forms that allow much finer control over how importing brings named values into your current module.

```
import module
from module import submodule
from . import submodule
```

We will discuss each of the import statement forms in turn starting with:

```
import module
```

This most basic type of import imports a module directly. Unlike Java, this form of import binds the left-most module name, so if you import a nested module like:

import greet.hello

you need to refer to it as 'greet.hello' and not just 'hello' in your code.

import greet.hello as foo

The 'as foo' part of the import allows you to relabel the 'greet.hello' module as 'foo' to make it more convenient to call. The example program uses this method to relabel 'greet.hello' as 'hello.' Note that it is not important that 'hello' was the name of the subpackage except that it might aid in reading the code. You would also use this technique if the identifier of the thing you wanted to import was already in use in this namespace: if you already had a variable called foo, and you wanted to import something else called foo, you could do import foo as bar.

From Import Statements

```
from module import name
```

This form of import allows you to import modules, classes or functions nested in other modules. This allows you to import code like this:

```
from greet import hello
```

In this case, it *is* important that 'hello' is actually a submodule of greet. This is not a relabeling but actually gets the submodule named 'hello' from the greet namespace. You can also use the from style of import to import all of the names in a module (except for those that start with an underscore) into your current module using a *. This form of import is discouraged in the Python community, and is particularly troublesome when importing from Java packages (in some cases it does not work) so you should avoid its use. It looks like this:

```
from module import *
```

If you are not importing from a Java package, it is sometimes convenient to use this form to pull in everything from another module.

Relative Import Statements

A new kind of import introduced in Python 2.5 is the explicit relative import. These import statements use dots to indicate how far back you will walk from the current nesting of modules, with one dot meaning the current module.

```
from . import module
from .. import module
from .module import submodule
from ..module import submodule
```

Even though this style of importing has just been introduced, its use is discouraged. Explicit relative imports are a reaction to the demand for implicit relative imports. If we had wanted to import the Greeter class out of greet.hello so that it could be instantiated with just Greeter() instead of greet.hello.Greeter we could have imported it like this:

```
from greet.hello import Greeter
```

If you wanted to import Greeter into the greet.people module, you could get away with:

```
from hello import Greeter
```

This is a relative import. Because greet.people is a sibling module of greet.hello, the 'greet' can be left out. This relative import style is deprecated and should not be used. Some developers like this style so that imports will survive module restructuring, but these relative imports can be error prone because of the possibility of name clashes. There is a new syntax that provides an explicit way to use relative imports, though they too are still discouraged. The previous import statement would look like this:

```
from .hello import Greeter
```

Aliasing Import Statements

Any of the above imports can add an 'as' clause to import a module but give it a new name.

```
import module as alias
from module import submodule as alias
from . import submodule as alias
```

This gives you enormous flexibility in your imports, so to go back to the greet.hello example, you could issue:

```
import greet.hello as foo
```

And use foo in place of greet.hello.

Hiding Module Names

Typically when a module is imported, all of the names in the module are available to the importing module. There are a couple of ways to hide these names from importing modules. Starting any name with an underscore (_) will document these names as private. The names are still accessible, they are just not imported when you import the names of a module with 'from module import *'. The second way to hide module names is to define a list named __all__, which should contain only those names that you wish to have your module to expose. As an example here is the value of __all__ at the top of Jython's OS module:

```
__all__ = ["altsep", "curdir", "pardir", "sep", "pathsep",
               "linesep", "defpath", "name", "path",
               "SEEK_SET", "SEEK_CUR", "SEEK_END"]
```

Note that you can add to __all__ inside of a module to expand the exposed names of that module. In fact, the os module in Jython does just this to conditionally expose names based on the operating system that Jython is running on.

Module Search Path, Compilation, and Loading

Understanding Jython's process of locating, compiling, and loading packages and modules is very helpful in getting a deeper understanding of how things really work in Jython.

Java Import Example

We'll start with a Java class which is on the CLASSPATH when Jython is started:

```
package com.foo;
public class HelloWorld {
    public void hello() {
        System.out.println("Hello World!");
    }
    public void hello(String name) {
        System.out.printf("Hello %s!", name);
    }
}
```

Here we manipulate that class from the Jython interactive interpreter:

```
>>> from com.foo import HelloWorld
>>> h = HelloWorld()
>>> h.hello()
Hello World!
>>> h.hello("frank")
Hello frank!
```

It's important to note that, because the HelloWorld program is located on the Java CLASSPATH, it did not go through the sys.path process we talked about before. In this case the Java class gets loaded directly by the ClassLoader. Discussions of Java ClassLoaders are beyond the scope of this book. To read more about ClassLoader see execution section of the Java language specification:

http://java.sun.com/docs/books/jls/second_edition/html/execution.doc.html.

Module Search Path and Loading

Understanding the process of module search and loading is more complicated in Jython than in either CPython or Java, because Jython can search both Java's CLASSPATH and Python's path. We'll start by looking at Python's path and sys.path. When you issue an import, sys.path defines the path that Jython will use to search for the name you are trying to import. The objects within the sys.path list tell Jython where to search for modules. Most of these objects point to directories, but there are a few special items that can be in sys.path for Jython that are not just pointers to directories. Trying to import a file that does not reside anywhere in the sys.path (and also cannot be found in the CLASSPATH) raises an ImportError exception. Let's fire up a command line and look at sys.path.

```
>>> import sys
>>> sys.path
['', '/Users/frank/jython/Lib', '__classpath__', '__pyclasspath__/',
'/Users/frank/jython/Lib/site-packages']
```

The first blank entry ('') tells Jython to look in the current directory for modules. The second entry points to Jython's Lib directory that contains the core Jython modules. The third and fourth entries are special markers that we will discuss later, and the last points to the site-packages directory where new libraries can be installed when you issue setuptools directives from Jython (see Appendix A for more about setuptools). The module that gets imported is the first one that is found along this path. Once a module is found, no more searching is done.

```
>>> import sys
>>> sys.path.append("/Users/frank/lib/mysql-connector-java-5.1.6.jar")
>>> import com.mysql
*sys-package-mgr*: processing new jar,
'/Users/frank/lib/mysql-connector-java-5.1.6.jar'
>>> dir(com.mysql)
['__name__', 'jdbc']
```

In this example, we added the mysql jar to the sys path, then when we tried to find com.mysql, the jar was scanned. Note that 'com.mysql' is a Java package that is found in mysql-connector-java-5.1.6.jar.

Java Package Scanning

Although you can ask the Java SDK to give you a list of all of the packages known to a ClassLoader using:

```
java.lang.ClassLoader#getPackages()
```

there is no corresponding

```
java.lang.Package#getClasses()
```

This is unfortunate for Jython, because Jython users expect to be able to introspect the code they use in powerful ways. For example, users expect to be able to call dir() on Java packages to see what they contain:

```
>>> import java.util.zip
>>> dir(java.util.zip)
['Adler32', 'CRC32', 'CheckedInputStream', 'CheckedOutputStream',
'Checksum', 'DataFormatException', 'Deflater',
'DeflaterOutputStream', 'GZIPInputStream', 'GZIPOutputStream',
'Inflater', 'InflaterInputStream', 'ZipEntry', 'ZipException',
'ZipFile', 'ZipInputStream', 'ZipOutputStream', '__name__']
```

And the same can be done on Java classes to see what they contain:

```
>>> import java.util.zip
>>> dir(java.util.zip.ZipInputStream)
['__class__', '__delattr__', '__doc__', '__eq__',
'__getattribute__', '__hash__', '__init__',
'__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__str__', 'available', 'class', 'close', 'closeEntry',
'equals', 'getClass', 'getNextEntry', 'hashCode', 'mark',
'markSupported', 'nextEntry', 'notify', 'notifyAll', 'read',
'reset', 'skip', 'toString', 'wait']
```

Making this sort of introspection possible in the face of merged namespaces requires some major effort the first time that Jython is started (and when jars or classes are added to Jython's path at runtime). If you have ever run a new install of Jython before, you will recognize the evidence of this system at work:

```
*sys-package-mgr*: processing new jar, '/Users/frank/jython/jython.jar'
*sys-package-mgr*: processing new jar,
'/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Classes/classes.j
ar'
*sys-package-mgr*: processing new jar,
'/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Classes/ui.jar'
*sys-package-mgr*: processing new jar,
'/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Classes/laf.jar'
...
*sys-package-mgr*: processing new jar,
'/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/lib/ext/sunj
ce_provider.jar'
*sys-package-mgr*: processing new jar,
'/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/lib/ext/sunp
kcs11.jar'
```

This is Jython scanning all of the jar files that it can find to build an internal representation of the package and classes available on your JVM. This has the unfortunate side effect of making the first startup on a new Jython installation painfully slow.

How Jython Finds the Jars and Classes to Scan

There are two properties that Jython uses to find jars and classes. These settings can be given to Jython using commandline settings or the registry (see Appendix A). The two properties are:

```
python.packages.paths
python.packages.directories
```

These properties are comma separated lists of further registry entries that actually contain the values the scanner will use to build its listing. You probably should not change these properties. The properties that get pointed to by these properties are more interesting. The two that potentially make sense to manipulate are:

```
java.class.path
java.ext.dirs
```

For the java.class.path property, entries are separated as the classpath is separated on the operating system you are on (that is, ‘;’ on Windows and ‘:’ on most other systems). Each of these paths are checked for a .jar or .zip and if they have these suffixes they will be scanned.

For the java.ext.dirs property, entries are separated in the same manner as java.class.path, but these entries represent directories. These directories are searched for any files that end with .jar or .zip, and if any are found they are scanned.

To control the jars that are scanned, you need to set the values for these properties. There are a number of ways to set these property values, see Appendix A for more.

If you only use full class imports, you can skip the package scanning altogether. Set the system property python.cachedir.skip to true or (again) pass in your own postProperties to turn it off.

Compilation

Despite the popular belief that Jython is ‘interpreted, not compiled,’ in reality all Jython code is turned into Java bytecode before execution. This Java bytecode is not always saved to disk, but when you see Jython execute any code, even in an eval or an exec, you can be sure that bytecode is getting fed to the JVM. The sole exception to this that we know of is the experimental pycimport module that we will describe in the section on sys.meta_path below, which interprets CPython bytecodes instead of producing Java bytecodes.

Python Modules and Packages versus Java Packages

The basic semantics of importing Python modules and packages versus the semantics of importing Java packages into Jython differ in some important respects that need to be kept carefully in mind.

sys.path

When Jython tries to import a module, it will look in its sys.path in the manner described in the previous section until it finds one. If the module it finds represents a Python module or package, this import will display a ‘winner take all’ semantic. That is, the first Python module or package that gets imported blocks any other module or package that might subsequently get found on any lookups. This means that if you have a module foo that contains only a name bar early in the sys.path, and then another module also called foo that only contains a name baz, then executing ‘import foo’ will **only** give you foo.bar and not foo.baz.

This differs from the case when Jython is importing Java packages. If you have a Java package org.foo containing bar, and a Java package org.foo containing baz later in the path, executing ‘import org.foo’ will **merge** the two namespaces so that you will get both org.foo.bar and org.foo.baz.

Just as important to keep in mind, if there is a Python module or package of a particular name in your path that conflicts with a Java package in your path this will also have a winner-take-all effect. If the Java package is first in the path, then that name will be bound to the merged Java packages. If the Python module or package wins, no further searching will take place, so the Java packages with the clashing names will never be found.

Naming Python Modules and Packages

Developers coming from Java will often make the mistake of modeling their Jython package structure the same way that they model Java packages. **Do not do this**. The reverse url convention of Java is a great, we would even say a brilliant convention for Java. It works very well indeed in the world of Java where these namespaces are merged. In the Python world however, where modules and packages display the winner-take-all semantic, this is a disastrous way to organize your code.

If you adopt this style for Python, say you are coming from ‘acme.com,’ you would set up a package structure like ‘com.acme.’ If you try to use a library from your vendor xyz that is set up as ‘com.xyz,’ then the first of these on your path will take the ‘com’ namespace, and you will not be able to see the other set of packages.

Proper Python Naming

The Python convention is to keep namespaces as shallow as you can, and make your top level namespace reasonably unique, whether it is a module or a package. In the case of acme and company xyz, you might start your package structures with ‘acme’ and ‘xyz’ if you wanted to have these entire codebases under one namespace (not necessarily the right way to go — better to organize by product instead of by organization, as a general rule).

Note

There are at least two sets of names that are particularly bad choices for naming modules or packages in Jython. The first is any top level domain like org, com, net, us, name. The second is any of the domains that Java the language has reserved for its top level namespaces: java, javax.

Advanced Import Manipulation

This section describes some advanced tools for dealing with the internal machinery of imports. It is pretty advanced stuff that is rarely needed, but when you need it, you **really** need it.

Import Hooks

To understand the way that Jython imports Java classes you have to understand a bit about the Python import protocol. We won't get into every detail, for that you would want to look at PEP 302 http://www.python.org/dev/peps/pep-0302/.

Briefly, we first try any custom importers registered on sys.meta_path. If one of them is capable of importing the requested module, allow that importer to handle it. Next, we try each of the entries on sys.path. For each of these, we find the first hook registered on sys.path_hooks that can handle the path entry. If we find an import hook and it successfully imports the module, we stop. If this did not work, we try the builtin import logic. If that also fails, an ImportError is thrown. So let's look at Jython's path_hooks.

sys.path_hooks

```
>>> import sys
>>> sys.path_hooks
[<type 'org.python.core.JavaImporter'>, <type 'zipimport.zipimporter'>,
<type 'ClasspathPyImporter'>]
```

Each of these path_hooks entries specifies a path_hook that will attempt to import special files. JavaImporter, as its name implies, allows the dynamic loading of Java packages and classes that are specified at runtime. For example, if you want to include a jar at runtime you can execute the following code:

```
>>> import sys
>>> sys.path.append("mysql-connector-java-5.1.6.jar")
>>> import com.mysql
*sys-package-mgr*: processing new jar, 'mysqlconnector-java-5.1.6.jar'
>>> dir(com.mysql)
['__name__', 'jdbc']
```

Note how the package scanning gets kicked off when 'com.mysql' is imported, as evidenced by the line starting with *sys-package-mgr*. Upon import, the JavaImporter scanned the new jar and allowed the import to succeed.

sys.meta_path

Adding entries to sys.meta_path allows you to add import behaviors that will occur before any other import is attempted, even the default builtin importing behavior. This can be a very powerful tool, allowing you to do all sorts of interesting things. As an example, we will talk about an experimental module that ships with Jython 2.5. That module is pycimport. If you start up Jython and issue

```
>>> import pycimport
```

Jython will start scanning for .pyc files in your path and, if it finds one, it will use the .pyc file to load your module.pyc files. These are the files that CPython produces when it compiles Python source code. So, after you have imported pycimport (which adds a hook to sys.meta_path) then issue:

```
>>> import foo
```

Jython will scan your path for a file named foo.pyc, and if it finds one it will import the foo module using the CPython bytecodes. It does this by creating a special class that defines a find_module method that specifies how to load in a pyc file. This class is then added to the meta search path with the sys.meta_path.insert method. The find_module method calls into other parts of pycimport and looks for .pyc files. If it finds one, it knows how to parse and execute those files and adds the corresponding module to the runtime. Pretty cool, eh?

Summary

In this chapter, you have learned how to divide code up into modules to for the purpose of organization and reuse. We have learned how to write modules and packages, and how the Jython system interacts with Java classes and packages.

Source: http://www.jython.org/jythonbook/en/1.0/ModulesPackages.html