Scripting With Jython

In this chapter, we will look at scripting with Jython. For our purposes, we will define ‘scripting’ as the writing of small programs to help out with daily tasks. These tasks are things like deleting and creating directories, managing files and programs, and anything else that feels repetitive that you might be able to express as a small program. In practice, however, scripts can become so large that the line between a script and a full sized program can blur.

We’ll start with some very small examples to give you a feel for the sorts of tasks you might script from Jython. Then we’ll cover a medium-sized task to show the use of a few of these techniques together.

Getting the Arguments Passed to a Script

All but the simplest of scripts will take some arguments from the command line. We’ll start our look at scripting by printing out any args passed to our script.

Listing 9-1.

```
import sys

for arg in sys.argv:
    print arg
```

Let’s try running our script with some random arguments:

Listing 9-2.

```
$ jython args.py a b c 1 2 3
args.py
a
b
c
1
2
3
```

The first value in sys.argv is the name of the script itself (args.py), the rest is the items passed in on the command line.

Searching for a File

Many scripting tasks take the form of ‘find a bunch of files and do something with them.’ So let’s take a look at how you might find some files from a Jython program. We’ll start with a simple script that finds any files in your current directory that match a passed in string:

Listing 9-3.

```
import sys
import os

for f in os.listdir(sys.argv[1]):
    if f.find(sys.argv[2]) != -1:
        print f
```

At the top of the script, we import the sys and os modules. Then we run a for loop over the results of os.listdir(sys.argv[1]). The os module is a good place to look if you are trying to accomplish the sorts of tasks you might do on a command prompt, such as listing the files in a directory, deleting a file, renaming a file, and the like. The listdir function of the os module takes one argument: a string that will be used as the path. The entries of the directory on that path are returned as a list. In this case, if we run this in its own directory (by passing in '.' for the current directory), we see:

Listing 9-4.

```
$ ls
args.py
search.py
$ jython list.py . py
args.py
search.py
$ jython list.py . se
search.py
```

In the first call to list.py, we list all files that contain 'py', listing 'args.py' and 'search.py.' In the second call, we list all files that contain the string 'se', so only 'search.py' is listed.

The os module contains many useful functions and attributes that can be used to get information about your operating environment. Next we can open up a Jython prompt and try out a few os features:

Listing 9-5.

```
>>> import os
>>> os.getcwd()
'/Users/frank/Desktop/frank/hg/jythonbook~jython-book/src/chapter8'
>>> os.chdir("/Users/frank")
>>> os.getcwd()
'/Users/frank'
```

We just printed out our current working directory with os.getcwd(), changed our current working directory to '/Users/frank,' and then printed out the new directory with another call to os.getcwd(). It is important to note that the JVM does not expose the ability to actually change the current working directory of the process running Jython. For this reason, Jython keeps track of its own current working directory. As long as you stay within Jython's standard library, the current working directory will behave as you expect (it will be changed with os.chdir()). However, if you import Java library functions that depend on a current working directory, it will always reflect the directory that Jython was started in.

Listing 9-6.

```
>>> import os
>>> os.getcwd()
'/Users/frank/Desktop/frank/hg/jythonbook~jython-book/src/chapter8'
>>> from java.io import File
>>> f = File(".")
>>> for x in f.list():
... print x
...
args.py
search.py
>>> os.chdir("/Users/frank")
>>> os.getcwd()
'/Users/frank'
>>> os.listdir(".")
['Desktop', 'Documents', 'Downloads', 'Dropbox', 'Library', 'Movies',
'Music', 'Pictures', 'Public', 'Sites']
>>> g = File(".")
>>> for x in g.list():
...     print x
...
args.py
search.py
```

Quite a bit went on in that last example, we'll take it step by step. We imported os and printed the current working directory, which is chapter8. We imported the File class from java.io. We then printed the contents of '.' from the Java side of the world. We then changed directories with os.chdir() to the home directory, and listed the contents of '.' from Jython's perspective, and listed '.' from the Java perspective. The important thing to note is that '.' from Java will always see the chapter8 directory because we cannot change the real working directory of the Java process—we can only keep track of a working directory so that Jython's working directory behaves the way Python programmers expect. Too many Python tools (like distutils and setuptools) depend on the ability to change working directories to ignore.

Manipulating Files

Listing files is great, but a more interesting scripting problem is actually doing something to the files you are working with. One of these problems that comes up for me from time to time is that of changing the extensions of a bunch of files. If you need to change one or two extensions, it isn't a big deal to do it manually. If you want to change hundreds of extensions, things get very tedious. Splitting extensions can be handled with the splitext function from the os.path module. The splitext function takes a file name and returns a tuple of the base name of the file and the extension.

Listing 9-7.

```
>>> import os
>>> for f in os.listdir("."):
... print os.path.splitext(f)
...
('args', '.py')
('builder', '.py')

('HelloWorld', '.java')
('search', '.py')
```

Now that we can get the extensions, we just need to be able to rename the files. Luckily, the os module has exactly the function we need, rename:

Listing 9-8.

```
>>> import os
>>> os.rename('HelloWorld.java', 'HelloWorld.foo')
>>> os.listdir('.')
['args.py', 'builder.py', 'HelloWorld.foo', 'search.py']
```

If you are manipulating any important files, be sure to put the names back!

```
>>> os.rename('HelloWorld.foo', 'HelloWorld.java')
>>> os.listdir('.')
['args.py', 'builder.py', 'HelloWorld.java', 'search.py']
```

Now that you know how to get extensions and how to rename files, we can put them together into a script (chext.py) that changes extensions:

Listing 9-9.

```
import sys
import os

for f in os.listdir(sys.argv[1]):
    base, ext = os.path.splitext(f)
    if ext[1:] == sys.argv[2]:
        os.rename(f, "%s.%s" % (base, sys.argv[3]))
```

Making a Script a Module

If you wanted to turn chext.py into a module that could also be used from other modules, you could put this code into a function and separate its use as a script like this:

Listing-9-10.

```
import sys
import os

def change_ext(directory, old_ext, new_ext):
    for f in os.listdir(sys.argv[1]):
        base, ext = os.path.splitext(f)
        if ext[1:] == sys.argv[2]:
            os.rename(f, "%s.%s" % (base, sys.argv[3]))

if __name__ == '__main__':
    if len(sys.argv) < 4:
        print "usage: %s directory old_ext new_ext" % sys.argv[0]
        sys.exit(1)
    change_ext(sys.argv[1], sys.argv[2], sys.argv[3])
```

This new version can be used from an external module like this:

Listing 9-11.

```
import chext

chext.change_ext(".", "foo", "java")
```

We have also used this change to introduce a little error checking, if we haven't supplied enough arguments, the script prints out a usage message.

Parsing Commandline Options

Many scripts are simple one-offs that you write once, use, and forget. Others become part of your weekly or even daily use over time. When you find that you are using a script over and over again, you often find it helpful to pass in command line options. There are three main ways that this is done in Jython. The first way is to hand parse the arguments that can be found from sys.argv as we did above in chext.py, the second is the getopt module, and the third is the newer, more flexible optparse module.

If you are going to do more than just feed the arguments to your script directly, then parsing these arguments by hand can get pretty tedious, and you'll be much better off using getopt or optparse. The optparse module is the newer, more flexible option, so we'll cover that one. The getopt module is still useful since it requires a little less code for simpler expected arguments. Here is a basic optparse script:

Listing 9-12.

```
# script foo3.py
from optparse import optionparser
parser = optionparser()
parser.add_option("-f", "--foo", help="set foo option")
parser.add_option("-b", "--bar", help="set bar option")
(options, args) = parser.parse_args()
```

```
print "options: %s" % options
print "args: %s" % args
```

running the above:

Listing 9-13.

```
$ jython foo3.py -b a --foo b c d
$ options: {'foo': 'b', 'bar': 'a'}
$ args: ['c', 'd']
```

In this example, we have created an optionparser and added two options with the add_option method. The add_option method takes at least one string as an option argument ('-f' in the first case) and an optional long version ('–foo' in the previous case). You can then pass in optional keyword options like the 'help' option that sets the help string that will be associated with the script. We'll come back to the optparse module with a more concrete example later in this chapter.

Compiling Java Source

While compiling Java source is not strictly a typical scripting task, it is a task that we'd like to show off in a bigger example starting in the next section. The API we are about to cover was introduced in jdk 6, and is optional for jvm vendors to implement. We know that it works on the jdk 6 from Sun (the most common JDK in use) and on the jdk 6 that ships with mac os x. For more details of the javacompiler api, a good starting point is here:http://java.sun.com/javase/6/docs/api/javax/tools/javacompiler.html.

The following is a simple example of the use of this API from Jython:

Listing 9-14.

```
from javax.tools import (ForwardingJavaFileManager, ToolProvider,
DiagnosticCollector,)
names = ["HelloWorld.java"]
compiler = ToolProvider.getSystemJavaCompiler()
diagnostics = DiagnosticCollector()
manager = compiler.getStandardFileManager(diagnostics, none, none)
units = manager.getJavaFileObjectsFromStrings(names)
comp_task = compiler.getTask(none, manager, diagnostics, none, none, units)
success = comp_task.call()
manager.close()
```

First we import some Java classes from the javax.tools package. Then we create a list containing just one string, 'HelloWorld.java.' Then we get a handle on the system Java compiler and call it 'compiler.' A couple of objects that need to get passed to our compiler task, 'diagnostics' and 'manager' are created. We turn our list of strings into 'units' and finally we create a compiler task and execute its call method. If we wanted to do this often, we'd probably want to roll up all of this into a simple method.

Example Script: Builder.py

So we've discussed a few of the modules that tend to come in handy when writing scripts for Jython. Now we'll put together a simple script to show off what can be done. We've chosen to write a script that will help handle the compilation of java files to .class files in a directory, and clean the directory of .class files as a separate task. We will want to be able to create a directory structure, delete the directory structure for a clean build, and of course compile our java source files.

Listing 9-15.

```
import os
import sys
import glob

from javax.tools import (forwardingjavafilemanager, toolprovider,
         diagnosticcollector,)

tasks = {}

def task(func):
    tasks[func.func_name] = func

@task
def clean():
    files = glob.glob("\*.class")
    for file in files:
        os.unlink(file)

@task
def compile():
    files = glob.glob("\*.java")
    _log("compiling %s" % files)
    if not _compile(files):
        quit()
    _log("compiled")

def _log(message):
    if options.verbose:
        print message

def _compile(names):
    compiler = toolprovider.getsystemjavacompiler()
    diagnostics = diagnosticcollector()
    manager = compiler.getstandardfilemanager(diagnostics, none, none)
    units = manager.getjavafileobjectsfromstrings(names)
    comp_task = compiler.gettask(none, manager, diagnostics, none, none,
units)
    success = comp_task.call()
    manager.close()
    return success

if __name__ == '__main__':
    from optparse import optionparser
    parser = optionparser()
    parser.add_option("-q", "--quiet",
        action="store_false", dest="verbose", default=true,
        help="don't print out task messages.")
    parser.add_option("-p", "--projecthelp",
        action="store_true", dest="projecthelp",
        help="print out list of tasks.")
    (options, args) = parser.parse_args()

    if options.projecthelp:
        for task in tasks:
            print task
        sys.exit(0)

    if len(args) < 1:
        print "usage: jython builder.py [options] task"
        sys.exit(1)

    try:
        current = tasks[args[0]]
    except KeyError:
        print "task %s not defined." % args[0]
        sys.exit(1)
    current()
```

The script defines a ‘task’ decorator that gathers the names of the functions and puts them in a dictionary. We have an optionparser class that defines two options –projecthelp and –quiet. By default the script logs its actions to standard out. The option –quiet turns this logging off, and –projecthelp lists the available tasks. We have defined two tasks, ‘compile’ and ‘clean.’ The ‘compile’ task globs for all of the .java files in your directory and compiles them. The ‘clean’ task globs for all of the .class files in your directory and deletes them. Do be careful! The .class files are deleted without prompting!

So let’s give it a try. If you create a Java class in the same directory as builder.py, say the classic ‘Hello World’ program:

HelloWorld.java

Listing 9-16.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

You could then issue these commands to builder.py with these results:

Listing 9-17.

```
[frank@pacman chapter8]$ jython builder.py --help
Usage: builder.py [options]

Options:
    -h, --help show this help message and exit
    -q, --quiet Don't print out task messages.
    -p, --projecthelp Print out list of tasks.
[frank@pacman chapter8]$ jython builder.py --projecthelp
compile
clean
[frank@pacman chapter8]$ jython builder.py compile
compiling ['HelloWorld.java']
compiled
[frank@pacman chapter8]$ ls
HelloWorld.java HelloWorld.class builder.py
[frank@pacman chapter8]$ jython builder.py clean
[frank@pacman chapter8]$ ls
HelloWorld.java builder.py
[frank@pacman chapter8]$ jython builder.py --quiet compile
[frank@pacman chapter8]$ ls
HelloWorld.class HelloWorld.java builder.py
[frank@pacman chapter8]$
```

Summary

This chapter has shown how to create scripts with Jython. We have gone from the most simple one- and two-line scripts to large scripts with lots of optional inputs.

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