Python: -m Is for Main

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly Media, 2013). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

As Python programmers know, Python doesn't really have a notion of a main() function like compiled languages such as C or Java. That is, there's no dedicated function that you define as the entry point to your program. Instead, there is the concept of a "main" program module. The "main" module holds the contents of whatever file you tell Python to execute. For example, if you type this,

```
bash % python spam.py
```

then the contents of spam.py become the main module. For scripts, you might also see the classic #! convention used to make them executable:

```
#!/usr/bin/env python
# spam.py
...
```

Finally, a common idiom found in most code meant to run as a main program is a check that looks like this:

```
# spam.py
...
if __name__ == '__main__':
    # Main program
    ...
```

__name__ is a special variable that always holds the name of the module and is set to '__main__' when executing as a main program. The primary reason for enclosing the main program in such a check is that it allows you to import the file as a library module without triggering main program execution. This can be useful for debugging, writing unit tests, etc.

For many programmers, this is the final word when it comes to writing scripts. I'll admit that for most of the past 15 years, I've never done much more than this or given the idea of a main script much thought. Naturally, there is more than meets the eye, otherwise, I wouldn't be writing about it. Let's dig a bit deeper.

The -m Option

Normally when you run a program, you simply give Python the name of the file that you want to execute; however, a less obvious way to specify the file is as a module name using -m. For example:

```
bash % python -m spam
```

Unlike a simple file name, the useful feature of -m is that it searches for spam on the Python path (sys.path). Although this feature is a minor change, it means that you don't actually have to know where spam.py is located to run it—spam.py merely must be located somewhere where Python can import it.

Once you discover -m, you'll quickly find that there is a wide range of built-in modules and tools that execute in this way. For example, if you want to run a simple Web server on a directory of files, do this:

```
bash % python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

If you want to run a program under the debugger, type this:

```
bash % python -m pdb yourprogram.py
```

Or to profile a program:

```
bash % python -m cProfile yourprogram.py
```

Or to time the execution of simple statements:

```
bash % python -m timeit --setup="import math" "math.cos(2)"
10000000 loops, best of 3: 0.125 usec per loop
bash %
```

Indeed, you'll find that there are a lot of useful things that live behind the -m option. Your application can use it, too. As it turns out, there are several benefits to doing so.

Organizing Large Applications

Almost any non-trivial Python program consists of both library modules and application-level scripts. When you're starting out, putting all of your code in a single directory and not worrying too much about code organization is often fine; however, as things start to grow, you'll want to think about having a better organization than a simple directory with a bunch of files in it. This is especially so if you're going to start giving your code away to others.

For most projects, putting library modules into a package structure is standard practice. You pick a unique top-level name for your project and organize code as a hierarchy. For example, if the name of your project was "diddy," you might make a directory like this:

```
diddy/
    __init__.py
    foo.py
    bar.py
    ...
```

If you've never seen __init__.py before, it's required to mark a directory as being part of a package. The file can be empty, but it must be there for imports to work. Application scripts would then be written to import modules out of this package using statements such as this:

```
# rundiddy.py
# An application script
from diddy import foo
from diddy import bar
...

if __name__ == '__main__':
    # Main program
    ...
```

This approach immediately presents some problems, though. In order for a script like this to work, the related package needs to be properly installed on the Python path (sys.path). This might not be a problem if you're working by yourself, but if you hand the script to a co-worker, it's not going to work unless she also has the associated libraries installed somewhere. As an alternative, you might consider putting the script in a common location (e.g., /usr/local/bin) and telling your co-workers to use that; however, you've now placed yourself in the role of a system administrator as you try to manage the script, the installed libraries, and everything else associated with your application.

All of these problems are caused by the fact that the script and its dependent package are placed in separate locations. As such, you need to worry about path settings, version dependencies, and all sorts of other installation issues. For example, how do you make sure that your script actually uses the right version of its dependent library package? I rarely run into Python coders who haven't ended up creating a big sys.path hacking mess for themselves trying to deal with things like this at one point or another; it can also cause all sorts of weird problems during code development. For example, you're probably not going to get the last few hours of debugging back after you realize that the reason your code is failing is that it was importing a version of a library different from the one you expected.

In-Package Scripts

One nice feature of the -m option is that it allows you to easily create "in-package" scripts. These are scripts that live in the same package hierarchy as the library files on which they rely. For example, you can simply move the rundiddy.py file inside the package like this:

```
diddy/
    __init__.py
    foo.py
    bar.py
    rundiddy.py
    ...
```

Once a script lives in a package, you can additionally modify it to use package-relative imports like this:

```
# rundiddy.py
# An application script
from . import foo
from . import bar
...
```

If you've never seen a package-relative import before, the syntax from . import foo means load foo from the same directory. Similarly, a statement such as from .. import foo loads a module from the parent directory whereas from ..utils import foo loads a module from the directory ../util relative to the module doing the import. I must stress that this syntax only works within a proper package—you can't use it in arbitrary Python modules. Additionally, you're not allowed to write an import that "escapes" the top-level package directory.

One nice thing about package-relative imports is that you no longer need to hard-code the top-level package name into the source, meaning that renaming the top-level package to something else is easy. For example, if you need to have two different versions of your package installed at the same time, rename one of them (e.g., "olddiddy"). All of the imports within the package will still work if they've been written using the package-relative style.

To run an in-package script, you simply type python -m diddy.rundiddy. If you've done things correctly, the script will simply find all of its correct library files, with no path hacking or installation headaches.

If you're put off by having to type python -m diddy.rundiddy, you can change the name of the rundiddy.py file to __main__.py. You'll then be able to type python -m diddy and it will simply run the __main__.py file for you. (As an aside, few programmers realize that any directory of code with a __main__.py file can be directly executed by Python.)

Who Cares?

The main benefit of moving scripts inside a package is that they effectively allow you to create a kind of code bundle where everything is self-contained. For example, if you wanted to give your application to a co-worker, you could simply hand them the top-level directory along with instructions on how to run the code (using -m). If you've done everything right, the code will simply "work" without ever having to fiddle with path settings, installing code into the user's Python installation, or anything else. During software development, this is actually a really useful thing—you can hand someone your code and have him try it out without requiring him to muck around with his local Python setup. Similarly, if you're working on a new version of code, you can do it in your own directory without ever worrying about previously installed versions getting in the way. Again, the key thing that makes this possible is the fact that everything is bundled together in one place.

I've found this approach to be useful in writing various application-level tools. For example, consider this hypothetical application structure:

```
diddy/
    __init__.py
    foo.py
    bar.py
    __main__.py
    server/
        __init__.py
        httpserver.py
        rpc.py
        message.py
        __main__.py
    worker/
        __init__.py
        queues.py
        request.py
        __main__.py
```

Within this directory, there are actually three separate "applications" that are executed using -m. For example:

```
bash % python -m diddy          # Executes diddy/__main__.py

bash % python -m diddy.server    # Executes diddy/server/__
main__.py

bash % python -m diddy.worker    # Executes diddy/worker/__
main__.py
```

Again, it's a self-contained bundle of code. There are no scripts to install and no path hacking to be had other than making sure the top level "diddy" directory is available when you run Python (it could be in the current working directory).

Application to Testing

Another place where I've found the package approach to be useful is in unit testing. A problem I always seem to face is figuring out how to make my unit tests use the correct version of code. That might sound silly, but I can't count the number of times I've run some tests only to find out that they executed using a completely different version of the code than the one I was working on due to some kind of sys.path issue. In response to such problems, you might be inclined to hack sys.path in some manner. For example, in one of my projects, if you look at the testing files, the first thing the tests do is hack sys.path to make sure the tests run using the right code base. Frankly, it's clumsy and a bit embarrassing.

As an alternative, you can move the tests inside the package and use the -m option to run them. For example, consider a project with this file structure:

```
diddy/
    __init__.py
    foo.py
    bar.py
    tests/
        __init__.py
        foo.py
        bar.py
        __main__.py
```

In this organization, the tests directory mirrors the structure of the package itself. Each testing file is a stand-alone executable that looks like this:

```
# tests/foo.py
import unittest
from .. import foo

class TestSomething(unittest.TestCase):
    def test_example(self):
        result = foo.do_something()
        self.assertEqual(result, expected_result)

    # More tests follow
    ...

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

To run a single testing file, you simply type a command like this:

```
bash % python -m diddy.tests.foo
.....
----------------------------------------------------------------------
Ran 5 tests in 0.295s

OK
bash %
```

I might reserve the tests/__main__.py for running all of the tests at once. For example, a simple approach is as follows:

```
# tests/__main__.py
from .foo import *
from .bar import *

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

Now, tests can be run like this:

```
bash % python -m diddy.tests
.........
----------------------------------------------------------------------
Ran 9 tests in 0.423s

OK
bash %
```

Saying whether such approach would appeal to hard-core testing experts is difficult; some might argue that the tests should be contained in their own dedicated directory separate from the package itself. To be sure, this might not scale for a tremendously huge project. Nevertheless, I've often found this approach to be simple, reliable, and quite effective in medium-scale projects. Part of the appeal is that it works without having to fiddle around with the environment or a complex set of extra tools. Of course, your mileage might vary.

Closing Words

Every so often a feature of Python comes along that really catches my fancy. The -m option definitely falls into that category as I find myself using it more and more. Honestly, the main appeal of it is how it allows my scripts and library code to be bundled together into a single cohesive package. As such, it saves me a lot of time where I would have to be fiddling around with path settings and installation issues. No, life is too short for that. Instead, put everything in a package and use -m. You'll thank yourself later.