Python: Import Anything

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also a co-author of the forthcoming *Python Cookbook* (3rd Edition, O'Reilly & Associates, 2013). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

In the August 2012 issue of *;login:*, I explored some of the inner workings of Python's import statement. Much of that article explored the mechanism that's used to set up the module search path found in sys.path as well as the structure of a typical Python installation. At the end of that article, I promised that there is even more going on with `import` than meets the eye. So, without further delay, that's the topic of this month's article.

Just as a note, this article assumes the use of Python 2.7. Also, because of the advanced nature of the material, I encourage you to follow along with the interactive examples as they nicely illustrate the mechanics of it all.

Import Revisited

Just to revisit a few basics, each Python source file that you create is a module that can be loaded with the import statement. To make the import work, you simply need to make sure that your code can be found on the module search path `sys.path`. Typically, `sys.path` looks something like this:

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python2.7/site-packages/setuptools-0.6c11-py2.7.egg',
 '/usr/local/lib/python2.7/site-packages/pip-1.1-py2.7.egg',
 '/usr/local/lib/python2.7/site-packages/python_dateutil-1.5-py2.7.egg',
 '/usr/local/lib/python2.7/site-packages/pandas-0.7.3-py2.7-macosx-
10.4-x86_64.egg',
 '/usr/local/lib/python2.7/site-packages/tornado-2.1-py2.7.egg',
 '/usr/local/lib/python27.zip',
 '/usr/local/lib/python2.7',
 '/usr/local/lib/python2.7/plat-darwin',
 '/usr/local/lib/python2.7/plat-mac',
 '/usr/local/lib/python2.7/plat-mac/lib-scriptpackages',
 '/usr/local/lib/python2.7/lib-tk',
 '/usr/local/lib/python2.7/lib-old',
 '/usr/local/lib/python2.7/lib-dynload',
 '/Users/beazley/.local/lib/python2.7/site-packages',
 '/usr/local/lib/python2.7/site-packages']
>>>
```

For most Python programmers (including myself until recently), knowledge of the `import` statement doesn't extend far beyond knowing about the path and the fact that it sometimes needs to be tweaked if code is placed in an unusual location.

Making Modules Yourself

Although most modules are loaded via `import`, you can actually create module objects yourself. Here is a simple interactive example you can try just to illustrate:

```
>>> import imp
>>> mod = imp.new_module("mycode")
>>> mod.__file__ = 'interactive'
>>> code = '''
... def hello(name):
...     print "Hello", name
...
... def add(x,y):
...     return x+y
... '''
>>> exec(code, mod.__dict__)
>>> mod
<module 'mycode' from 'interactive'>
>>> dir(mod)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'add',
'hello']
>>> mod.hello('Dave')
Hello Dave
>>> mod.add(10,20)
30
>>>
```

Essentially, if you want to make a module you simply use the `imp.new_module()` function. To populate it, use the `exec` statement to execute the code you want in the module.

As a practical matter, the fact that you can make modules from scratch (bypassing import) may be nothing more than a curiosity; however, it opens a new line of thought. Perhaps you could create modules in an entirely different manner than a normal import statement, such as grabbing code from databases, from remote machines, or different kinds of archive formats. What's more, if all of this is possible, perhaps there is some way to customize the behavior of import directly.

Creating an Import Hook

Starting around Python 2.6 or so, the sys module acquired a mysterious new variable `sys.meta_path`. Initially, it is set to an empty list:

```
>>> import sys
>>> sys.meta_path
[]
>>>
```

What purpose could this possibly serve? To find out, try the following experiment:

```
>>> class Finder(object):
...     def find_module(self, fullname, path=None):
...         print "Looking for", fullname, path
...         return None
...
>>> import sys
>>> sys.meta_path.append(Finder())
>>> import math
Looking for math None
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml._xmlplus ['/usr/local/lib/python2.7/xml']
Looking for _xmlplus None
Looking for xml.etree ['/usr/local/lib/python2.7/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.sys ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.re ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.warnings ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.ElementPath ['/usr/local/lib/python2.7/xml/etree']
Looking for xml.etree.ElementC14N ['/usr/local/lib/python2.7/xml/etree']
Looking for ElementC14N None
>>>
```

Wow, look at that! The `find_module()` method of the Finder class you just wrote is suddenly being triggered on every single import statement. As input, it receives the fully qualified name of the module being imported. If the module is part of a package, the path argument is set to the package's `__path__` variable, which is typically a list of subdirectories that contain the package subcomponents. With packages, there are also a few unexpected oddities. For example, notice the attempted imports of `xml.etree.sys` and `xml.etree.re`. These are actually imports of `sys` and `re` occurring inside the `xml.etree` package. (Later these are tested for a relative and then absolute import.)

As output, the `find_module()` either returns `None` to indicate that the module isn't known or returns an instance of a loader object that will carry out the process of loading the module and creating a module object. A loader is simply some object that defines a `load_module` method that returns a module object created in a manner as shown earlier. Here is an example that mirrors the creation of the module that was used earlier:

```
>>> import imp
>>> import sys
>>> class Loader(object):
...     def load_module(self, fullname):
...             mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
...             code = '''
... def hello(name):
...     print "Hello", name
...
... def add(x,y):
...     return x+y
... '''
...             exec(code, mod.__dict__)
...             return mod
```

```
...
>>> class Finder(object):
...     def find_module(self, fullname, path):
...           if fullname == 'mycode':
...                 return Loader()
...           else:
...                 return None
...
>>> sys.meta_path.append(Finder())
>>> import mycode
>>> mycode.hello('Dave')
Hello Dave
>>> mycode.add(2,3)
5
>>>
```

In this example, the code is mostly straightforward. The `Finder` class creates a `Loader` instance. The loader, in turn, is responsible for creating the module object and executing the underlying source code. The only part that warrants some discussion is the use of `sys.modules.setdefault()`. The `sys.modules` variable is a cache of already loaded modules. Updating this cache as appropriate during import is the responsibility of the loader. The `setdefault()` method makes sure that this happens cleanly by either returning the module already present or a new module created by `imp.new_module()` if needed.

Using Import Hooks

Defining an import hook opens up a variety of new programming techniques. For instance, here is a finder that forbids imports of certain modules:

```
# forbidden.py

import sys

class ForbiddenFinder(object):
   def __init__(self, blacklist):
      self._blacklist = blacklist

   def find_module(self, fullname, path):
      if fullname in self._blacklist:
         raise ImportError()

def no_import(module_names):
   sys.meta_path.append(ForbiddenFinder(module_names))
```

Try it out:

```
>>> import forbidden
>>> forbidden.no_import(['xml','threading','socket'])
>>> import xml
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named xml
>>> import threading
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named threading
>>>
```

Here is a more advanced example that allows callback functions to be attached to the import of user-specified modules:

```
# postimport.py

import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path):
        print "Finding", fullname, path
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        try:
            importlib.import_module(fullname)
            modname = fullname
        except ImportError:
            package, _, modname = fullname.rpartition('.')
            if package:
                try:
                    importlib.import_module(modname)
                except ImportError:
                    return None
            else:
                return None
        module = sys.modules[modname]
        for func in _post_import_hooks[modname]:
            func(module)
        _post_import_hooks[modname] = []
        self._finder._skip.remove(fullname)
        return module
```

```
def on_import(modname, callback):
    if modname in sys.modules:
        callback(sys.modules[modname])
    else:
        _post_import_hooks[modname].append(callback)

sys.meta_path.insert(0, PostImportFinder())
```

The idea on this hook is that it gets triggered on each import; however, immediately upon firing, it disables itself from further use. The `load_module()` method in the `PostImportLoader` class then carries out the regular import and triggers the registered callback functions. There is a bit of a mess concerning attempts to import the requested module manually. If an attempt to import the fully qualified name doesn't work, a second attempt is made to import just the base name.

To see it in action, try the following:

```
>>> from postimport import on_import
>>> def loaded(mod):
...     print "Loaded", mod
...
>>> on_import('math', loaded)
>>> on_import('threading', loaded)
>>> import math
Loaded <module 'math' from '/usr/local/lib/python2.7/lib-dynload/math.so'>
>>> import threading
Loaded <module 'threading' from '/usr/local/lib/python2.7/threading.pyc'>
>>>
```

Although a simple example has been shown, you could certainly do something more advanced such as patch the module contents. Consider this additional code that adds logging to selected functions:

```
def add_logging(func):
    'Decorator that adds logging to a function'
    def wrapper(*args, **kwargs):
        print("Calling %s.%s" % (func.__module__, func.__name__))
        return func(*args, **kwargs)
    return wrapper

def log_on_import(qualified_name):
    'Apply logging decorator to a function upon import`
    modname, _, symbol = qualified_name.rpartition('.')
    def patch_module(mod):
        setattr(mod, symbol, add_logging(getattr(mod, symbol)))
    on_import(modname, patch_module)
```

Here is an example:

```
>>> from postimport import log_on_import
>>> log_on_import('math.tan')
>>>
>>> import math
>>> math.tan(2)
Calling math.tan
```

```
-2.185039863261519
>>>
```

You might look at something like this with horror; however, you could also view it as a way to manipulate a large code base without ever touching its source code directly. For example, you could use an import hook to insert probes, selectively rewrite part of the code, or perform other actions on the side.

Path-Based Hooks

Manipulation of `sys.meta_path` is not the only way to hook into the import statement. As it turns out, there is another variable `sys.path_hooks` that can be manipulated. Take a look at it:

```
>>> import sys
>>> sys.path_hooks
[<type 'zipimport.zipimporter'>]
>>>
```

The items on `sys.path_hooks` are callables that process individual items in the `sys.path` list, and it either responds with an ImportError or it returns a finder object that is used to load modules from that path component. Try this experiment:

```
>>> import sys
>>> def check_path(name):
...     print "Checking", repr(name)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
>>> # Clear the cache to have all path entries rechecked
>>> sys.path_importer_cache.clear()
>>> import foo
Checking ''
Checking '/usr/local/lib/python27.zip'
Checking '/usr/local/lib/python2.7'
Checking '/usr/local/lib/python2.7/plat-darwin'
Checking '/usr/local/lib/python2.7/plat-mac'
Checking '/usr/local/lib/python2.7/plat-mac/lib-scriptpackages'
Checking '/usr/local/lib/python2.7/lib-tk'
Checking '/usr/local/lib/python2.7/lib-old'
Checking '/usr/local/lib/python2.7/lib-dynload'
Checking '/usr/local/lib/python2.7/site-packages'
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named foo
>>>
```

Notice how every entry on `sys.path` is checked by our function. To expand this code, you would make the `check_path()` function look for a specific pathname pattern. If found, it returns a special finder object that's similar to before. Try this:

```
>>> class Finder(object):
...     def find_module(self, name, path=None):
...         print "Looking for", name, path
...         return None
```

```
...
>>> def check_path(name):
...     if name.endswith('.spam'):
...         return Finder()
...     else:
...         raise ImportError()
...
>>> import sys
>>> sys.path_hooks.append(check_path)
>>> import foo
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named foo
>>> sys.path.append('code.spam')
>>> import foo
Looking for foo None              # Notice Finder output here
Traceback (most recent call last):
  File "<stdin>", line 1, in
ImportError: No module named foo
>>>
```

This technique of hooking into sys.path is how Python has been expanded to import from .zip files and other formats.

Final Words and the Big Picture

Hacking of Python's import statement has been around for quite some time, but it's often shrouded in magic and mystery. Frameworks and software development tools will sometimes do it to carry out advanced operations across an entire code base; however, the whole process is poorly documented and underspecified. For instance, internally, Python 2.7 doesn't use the same machinery as extensions to the import statement. Frankly, it's a huge mess.

One of the most significant changes in the recent Python 3.3 release is an almost complete rewrite and formalization of the import machinery described here. Internally, it now uses `sys.meta_path` and path hooks for all stages of the import process. As a result, it's much more customizable (and understandable) than previous versions.

Having seen of all of this, should you now start hacking on `import`? Probably not; however, if you want to have a deep understanding of how Python is put together and how to figure things out when they break, knowing a bit about it is useful. For more information about import hooks, see PEP 302, http://www.python.org/dev/peps/pep-0302/.