Import That!

DAVID BEAZLEY

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

I have a small confession: I often think that I don't fully understand the Python `import` statement. As a seasoned Python veteran, it might be surprising to admit something like that, but I think I might be in good company—most of the Python programmers I meet are just as mystified by some of the inner workings of it as I am.

Obviously, the `import` statement is one of the most important parts of Python. You use it to load standard library modules and third-party extensions. It's at the heart of various Python packaging and distribution tools. However, it's also the bane of anyone who needs to set up and maintain a custom Python installation (or anyone who has had to debug a broken setup).

In this article, I'm hoping to go behind the scenes and explain some of the mysteries surrounding module imports with the hope that it might be useful for anyone who has deal with setting up or debugging their Python installation.

Import Basics

As you know, the `import` statement is used to load library modules. For example:

```
import math
```

Modules define namespaces, so to access anything contained inside, you have to use the module name as a prefix. For example:

```
x = math.sin(2)
y = math.cos(2)
```

Alternatively, you can use the `from module import` form of import:

```
from math import sin, cos
```

This loads the module the same as before, but makes references to the listed symbols in the namespace of the code making the import. You can use those symbols without a prefix. For example:

```
x = sin(2)
y = cos(2)
```

Various details about *using* the `import` statement can be found in most Python books. That's not our focus here—instead, let's peel back the covers and go a bit deeper into its inner magic.

What Can Be Imported?

When you use a statement such as `import spam`, Python looks for one of three things. First, it checks for the existence of a package. A package is a directory with the same name as what is being imported and which contains an __init__.py file. On import, the __init__.py file executes.

If a package can't be found, the interpreter checks for a compiled C/C++ extension module in the form of a shared library or DLL. Depending on platform, this is typically a file such as spam.so (UNIX) or spam.pyd (Windows). Many of Python's standard library modules are actually compiled C code (math, time, etc.).

If a package directory or extension module can't be found, an attempt to load a simple Python source file is made. For this, Python first checks for a .pyc file containing compiled bytecode. If the format of the .pyc file matches the correct Python version and the timestamp is newer than the corresponding .py file, then the .pyc is loaded. Otherwise, the .py file is loaded and the matching .pyc file is overwritten with a more up-to-date version. It should be stressed that .pyc files are mostly a detail that you don't need to worry about. They are automatically created and managed by Python behind the scenes.

There is an underlying precedence for packages, extensions, and source files that happen to share the same name. For example, if you had a package spam and a source file spam.py, using `import spam` will always load the package and ignore the source file. Similarly, a C extension spam.so takes precedence over a source file spam.py.

It's All About the Path

To find packages, extensions, and modules, Python relies on the setting of the sys.path variable. You should launch Python on your machine and take a look at it:

```
>>> 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']
>>>
```

At a simple level, sys.path is simply a list of directory names. When you type `import spam`, Python starts scanning the list looking for the first match. That is,

each directory on the path is checked for a package, extension module, or simple source file that matches the import name.

Close inspection of sys.path reveals a bit more, however. For example, in addition to directories, you will often see .egg and .zip files. When a .zip file is added to the path, Python treats its contents as a logical directory. That is, matching files will be transparently loaded from a zip archive as if the .zip file had been unzipped into an actual directory. Files with an .egg type are typically associated with third-party libraries and add-on packages. An egg is either a directory or a .zip file in disguise, with some extra metadata stored inside. If you examine the contents of an egg, you'll find a directory `EGG-INFO` that has this information.

Almost all of the problems related to managing a complicated Python installation are due to issues concerning the setting of sys.path. For instance, if the `import` statement fails to load code, it means that code is located in a directory not listed on the path. If you are getting bizarre name clashes or Python is using the wrong version of a module, it also points to some issue concerning the path. If you tried to install a third-party package but the code doesn't work, you might have installed it into the wrong version of Python. The bottom line is that spending a few minutes to deconstruct the sys.path is a worthwhile experiment if you want to better understand how Python installs itself on your machine.

Adjusting the Path

The Python import system path is something that can be adjusted. Since sys.path is a list, you can simply change its contents to any set of directories that you wish. For example, you can write code that inserts new directories:

```
import sys
sys.path.insert(0,"/some/new/directory")
```

Even though you can do this, don't! In fact, writing code that manually tweaks the import path is the first step towards a future of endless installation sorrow. In part, you don't want to be writing code that directly messes around with hard-coded directories and file names like this (what if your code gets moved somewhere else?). Second, it can be hard to predict how your changes to the path will interact with other parts of Python (other libraries, frameworks, package managers, and so forth). Frankly, you're often best off leaving the path alone.

A somewhat more sane way to alter the path is to set the PYTHONPATH environment variable prior to running the interpreter. For example:

```
bash % env PYTHONPATH=/some/new/path:/another/path python
Python 2.7.1 (r271:86832, Feb 27 2011, 11:47:28)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 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',
```

```
  '/some/new/path',           # Note new paths added here
  '/another/path',
  '/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',
  '/usr/local/lib/python2.7/site-packages',
>>>
```

As you can see, setting PYTHONPATH causes new search directories to be added to the path. This is somewhat better than hard-coding such changes to sys.path in your code (as no source code modifications are needed). However, it's still a bit hard to predict where your new paths will be added to sys.path. For example, in the above example, why did the new directories get inserted into the middle of sys.path instead of the beginning or end? Mysteries abound.

Understanding Python's Installation

To better understand sys.path, it helps to step back and discuss a typical Python installation. Python typically installs itself in a directory such as /usr/local. Although this may vary, the top-level directory used by the interpreter can be found in the sys.prefix variable:

```
>>> import sys
>>> sys.prefix
  '/usr/local'
>>>
```

Under this top-level prefix, the Python interpreter itself is usually found in an executable sys.prefix + '/bin/python2.7' such as /usr/local/bin/python2.7. If you have multiple versions of Python installed, they are accessible using their respective version numbers such as python2.5, python2.6, or python3.2. The unversioned Python interpreter is often a link to one of these installed versions.

Python-related scripts are also installed in the same directory as the interpreter binary. For example, if you are using easy_install or pip to install third-party packages, those installer programs are found in /usr/local/bin alongside the interpreter. If multiple versions of Python have been installed, you actually have to be somewhat careful using such scripts. A common mistake is using a package installer to install things into the wrong version of the interpreter.

Python's standard library is located in a directory sys.prefix + '/lib/python2.7' such as /usr/local/lib/python2.7. To see the raw structure of the standard library, you can run Python with the `-S` option. This runs the interpreter, but in a mode that ignores third-party add-ons. You'll just get the bare interpreter and standard library. Try it:

```
bash % python -S
Python 2.7.3 (default, May 24 2012, 22:03:52)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
>>> import sys
```

```
>>> sys.path
['',
 '/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']
>>>
```

Within the library, modules and packages consisting of pure Python code are located in sys.prefix + '/lib/python2.7'. You should go take a look at it yourself. For example:

```
bash % ls /usr/local/lib/python2.7
... look at the output ...
```

In the output, you'll see a large number of hopefully recognizable files from the standard library.

C extension modules are located in sys.prefix + '/lib/python/lib-dynload'. Take a look at this as well:

```
bash % ls /usr/local/lib/python2.7/lib-dynload
... look at the output ...
```

Here, you'll see some of the lower-level system libraries related to sockets, timing, files, and so forth. All of the files are typically C shared libraries in the form of .so files.

Other parts of the library are a bit more special-purpose. For example, the plat-mac has some files containing platform-specific constants, and tib-tk contains some files related to the optional Tkinter (Tcl/Tk) package.

A Brief Word About Path Construction

The core part of sys.path is hardwired according to where Python is installed. When you run `python -S` as shown in the previous section, you get the basic path settings that Python starts with.

One subtle feature of the interpreter is that it can be "homed" to a different installation directory using an environment variable. For example:

```
bash % env PYTHONHOME=/my/python python -S
Python 2.7.3 (default, May 24 2012, 22:03:52)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
>>> import sys
>>> sys.path
['',
 '/my/python/lib/python27.zip',
 '/my/python/lib/python2.7/',
 '/my/python/lib/python2.7/plat-darwin',
 '/my/python/lib/python2.7/plat-mac',
 '/my/python/lib/python2.7/plat-mac/lib-scriptpackages',
```

```
  '/my/python/lib/python2.7/lib-tk',
  '/my/python/lib/python2.7/lib-old',
  '/my/python/lib/python2.7/lib-dynload']
>>>
```

The interpreter can also be re-homed through the use of "landmark" files. For example, when the interpreter starts, it determines its location on the filesystem and checks for the existence of ../lib/python2.7/os.py and ../lib/python2.7/lib-dynload. If those two files exist, Python assumes that you've relocated the installation and adjusts the path settings accordingly.

Of course, unless you've taken steps to set up the new directories as a proper installation, nothing much will work if you do this.

The PYTHONPATH environment variable specifies directories to be included just before the standard set of library directories. For example:

```
bash % env PYTHONPATH=/some/new/path:/another/path python -S
Python 2.7.3 (default, May 24 2012, 22:03:52)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
>>> import sys
>>> sys.path
['',
  '/some/new/path',
  '/another/path',
  '/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']
>>>
```

Third-Party Packages and Extensions

As Python programmers know, there are a huge number of third-party packages that can be added to Python. Within the Python standard library, there is a special directory dedicated to these extensions called site-packages (note: sometimes it's called dist-packages on certain Linux distributions). Typically, it is located at sys.prefix + '/lib/python2.7/site-packages'. Here is a partial listing of the site-packages directory of my machine:

```
bash % ls /usr/local/lib/python2.7/site-packages
IPython
certifi
chameleon
chardet
dateutil
easy-install.pth
mako
markupsafe
matplotlib
```

```
mpl_toolkits
numpy
oauthlib
pandas-0.7.3-py2.7-macosx-10.4-x86_64.egg
paste
pip-1.1-py2.7.egg
...
```

Handling of site packages is a little more complicated than it might initially seem. When Python starts, it imports a special module called site.py. One of the first things that happens is that site.py adds the site-packages directory to sys.path. You can often see it appended at the end of the normal library directories.

```
>>> import sys
>>> sys.path
['',
  ...
  '/usr/local/lib/python2.7/lib-dynload',
  '/Users/beazley/.local/lib/python2.7/site-packages # Added by site.py
  '/usr/local/lib/python2.7/site-packages',  # Added by site.py ]
>>>
```

You may also see a directory for user-level site packages being added as well. Since Python is often installed system-wide, it may not be possible for individual users to install Python packages, due to permissions. The user-level site package directory is reserved for this (each user can put custom packages in their per-user site-packages directory instead).

However, there's more to the management of packages than simply adding a few directories to the path. site.py also scans the site-packages directory for .pth files with additional path configuration. For example, here are some of the .pth files on my machine:

```
bash % cd /usr/local/lib/python2.7/site-packages
bash % ls *.pth
PasteDeploy-1.5.0-py2.7-nspkg.pth
easy-install.pth
repoze.lru-0.5-py2.7-nspkg.pth
setuptools.pth
zope.deprecation-3.5.1-py2.7-nspkg.pth
zope.interface-3.8.0-py2.7-nspkg.pth
```

These .pth files contain additional directories that must be added to the sys.path variable. For example, let's look at easy-install.pth:

```
bash % cat easy-install.pth
import sys; sys.__plen = len(sys.path) ./setuptools-0.6c11-py2.7.egg
./pip-1.1-py2.7.egg
./python_dateutil-1.5-py2.7.egg
./pandas-0.7.3-py2.7-macosx-10.4-x86_64.egg
./tornado-2.1-py2.7.egg
import sys; new=sys.path[sys.__plen:]; del sys.path[sys.__plen:];
p=getattr(sys,'__egginsert',0); sys.path[p:p]=new; sys.__egginsert = p+len(new)
```

In this file, lines starting with `import sys` are executed as Python code and can be seen to be doing things with sys.path. The other lines of the file list additional entries that need to be added to the path. It should be noted that the above code is a bit complicated due to the fact that it moves added directories from the end of sys.path to the beginning. This is the purpose of the rather cryptic line at the end.

If you want, you can make your own .pth files and put them in the site-packages directory. For example, here's a simple experiment you can try. Make a file called `mypath.pth` that contains the following:

```
# mypath.pth
/etc
/tmp
```

Now, copy this file to /usr/local/lib/python2.7/site-packages or ~/.local/lib/python2.7/site-packages and run Python. Here is an annotated listing of sys.path that shows the output and how the different parts are set:

```
bash % env PYTHONPATH=/some/new/path:/another/path python
Python 2.7.1 (r271:86832, Feb 27 2011, 11:47:28)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['',
# --- Added by site-packages/easy_install.pth files
'/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',
# --- Added by PYTHONPATH environment
 '/some/new/path',
'/another/path',
# --- Standard Python library (initialized at startup)
 '/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',
# --- Added by site.py
 '/usr/local/lib/python2.7/site-packages',
# --- Directories list in mypath.pth
 '/etc',
 '/tmp']
>>>
```

Don't forget to delete the mypath.pth file after you're done experimenting with it.

At this point, I would encourage you to dissect the sys.path setting on your own machine to see if you can figure out how it gets put together.

How to Use This Knowledge

Knowing how sys.path gets constructed is useful if you have to do any kind of maintenance on a Python setup. For instance, you'll know where third-party packages go (site-packages) and be able to diagnose common installation problems.

Custom Python distributions (e.g., Enthought Python Distribution, ActivePython) as well as vendor-supplied Python versions (e.g., those provided by Apple, Linux distributions, etc.) often involve customizations of the site.py library to adjust module import paths.

If you've ever wondered how various Python package management tools such as setuptools, distribute, or pip work, they also mostly involve manipulation of sys.path through the use of .pth files in site-packages. Additionally, they may make their own changes to site.py as well.

Other popular tools such as virtualenv also play games with the Python installation (see http://pypi.python.org/pypi/virtualenv). For example, with virtualenv, you can make isolated Python environments:

```
bash % virtualenv example
New python executable in example/bin/python
Installing setuptools.............done.
Installing pip...............done.
bash %
```

This creates a new Python installation under a directory "example." If you look at it, you'll see something that looks like a typical Python installation:

```
bash % cd example
bash % ls
bin    include lib
bash % ls bin
activate     activate_this.py pip
activate.csh   easy_install   pip-2.7
activate.fish    easy_install-2.7  python
bash % ls lib
python2.7
bash %
```

If you run the Python interpreter in this environment and inspect the path, you'll see how the path has been configured for an entirely new location, yet parts of the path incorporate directories from the original Python installation. Again, it's a bunch of clever path hacking.

```
bash % example/bin/python
Python 2.7.3 (default, May 24 2012, 22:03:52)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin Type "help", "copyright",
"credits" or "license" for more information.
>>> import sys
>>> sys.path
['',
```

```
'/Users/beazley/example/lib/python2.7/site-packages/setuptools-0.6c11-
py2.7.egg',
'/Users/beazley/example/lib/python2.7/site-packages/pip-1.1-py2.7.egg',
'/Users/beazley/example/lib/python27.zip',
'/Users/beazley/example/lib/python2.7',
'/Users/beazley/example/lib/python2.7/plat-darwin',
'/Users/beazley/example/lib/python2.7/plat-mac',
'/Users/beazley/example/lib/python2.7/plat-mac/lib-scriptpackages',
'/Users/beazley/example/lib/python2.7/lib-tk',
'/Users/beazley/example/lib/python2.7/lib-old',
'/Users/beazley/example/lib/python2.7/lib-dynload',
'/usr/local/lib/python2.7',
'/usr/local/lib/python2.7/plat-darwin',
'/usr/local/lib/python2.7/lib-tk',
'/usr/local/lib/python2.7/plat-mac',
'/usr/local/lib/python2.7/plat-mac/lib-scriptpackages',
'/Users/beazley/example/lib/python2.7/site-packages']
>>>
```

Final Words

Figuring out how Python installations are put together is often one of the most mysterious parts of Python for newcomers and even for experienced programmers. It's important to stress that if you've ever had problems importing libraries, it's invariably some kind of configuration issue related to sys.path.

Believe it or not, there are even more tricks that can be played with in handling the `import` statement. However, I'll leave that for a future article.