The Wheels Keep on Spinning

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. http://www.dabeaz.com, dave@dabeaz.com

If you've ever had the pleasure of installing third-party Python packages, you already know that it can be a bit of a mess. There are a variety of different tools, file formats, and other packaging complications. Frankly, it's enough to make your head spin.

Over the past year, a new Python packaging format has emerged in the form of a "wheel" file—so named because a wheel is a common packaging form factor for cheese, as in the big wheel of cheese that you might find at a cheese shop. Naturally, this is a reference to a certain cheese shop in an obscure Monty Python sketch, but that should have been obvious. I digress.

When the new wheel format emerged, I'll admit that I mostly ignored it. Python packaging is not my favorite topic, and the thought of having to think about yet another file format was relatively low on my list of day-to-day priorities; however, in recent months there has been a concerted effort to have package maintainers support the new wheel format. For example, the Web site http://pythonwheels.com/ currently shows the wheel status for the most popular Python extensions. As the author of one such extension, I was starting to get questions about wheels and was ashamed to admit my ignorance.

So, what in the heck is a wheel, you ask? In this installment, we'll take a look at wheels, Python packaging, and related topics. As we'll see, there are some rather interesting aspects to wheels—especially for anyone who needs to maintain, test, or deploy complex Python applications.

A Quick Review of Python Packaging

Before jumping into the subject of wheels, a quick refresher on Python packaging is probably in order. First, the Python Package Index (PyPI, at http://pypi.python.org) is the definitive site for locating and downloading third-party packages. If you go here, you'll find virtually all available packages listed, along with links to downloads, documentation, and more.

If you want to, you can download a package directly from PyPI and install it manually on your machine. Typically you would download the source and look for an enclosed setup.py file. You would then run `python setup.py install` on that file to perform an installation. For example, if you wanted to download the pytz extension for handling time zones, here are the steps that you might perform. In this example, replace the `curl` command with anything that simply downloads the source from PyPI. Also, depending on how Python has been installed, the final step might need to be performed as root using sudo.

```
bash % curl -O https://pypi.python.org/packages/source/p/pytz/pytz-2013.8.tar.gz
bash % tar -xf pytz-2013.8.tar.gz
bash % cd pytz-2013.8
bash % python setup.py install
```

Instead of manually installing a package in this manner, an alternative approach is to use the optional setuptools package. setuptools gives you the `easy_install` command, which automatically contacts PyPI, downloads the most recent version, and installs it for you. For example, instead of typing the above commands, you could simply type the following statement (again, you may need to use sudo depending on your Python installation):

```
bash % easy_install pytz
```

setuptools saves you the trouble of downloading, unpacking, and running the setup.py file yourself; however, it does quite a bit more than that because it will also download and install any dependencies. This can be useful if you're installing something much more complicated. For example, if you wanted to install the Python data analysis library pandas (http://pandas.pydata.org/), typing `easy_install pandas` will not only install pandas but also all of its dependencies, including numpy, python-dateutil, six, and pytz.

Although easy_install is commonly described in tutorials and documentation, the pip command (http://www.pip-installer.org) is a bit more modern, performs a similar function, and seems to be coming the preferred way to install packages. pip operates in a manner similar to easy_install. For example, to install a package, type a command like this:

```
bash % pip install pandas
```

Under the covers, pip actually requires the use of setuptools, so if you're using it, you'll actually have both easy_install and pip installed. This obviously begs the question: What is the major difference between the two? That's a big question, but pip makes a number of subtle changes to the installation process. For example, pip downloads all of the dependencies and builds them completely before attempting any kind of install. As a result, the install will either succeed in its entirety or not at all. On the other hand, easy_install might end up performing a partial install if some part of the installation process fails midway through. pip also provides some additional commands, such as the ability to uninstall a package.

Perhaps the most notable feature of pip is its ability to "freeze" and recreate your exact installation configuration. For example, suppose you had spent a lot of time making a custom Python setup. You can type the following command to freeze it into a requirements file:

```
bash % pip freeze >requirements.txt
```

This creates a file requirements.txt that looks like this:

```
Django==1.6
SQLAlchemy==0.8.3
numpy==1.8.0
pandas==0.12.0
ply==3.4
python-dateutil==2.2
pytz==2013.8
requests==2.0.1
six==1.4.1
virtualenv==1.10.1
wsgiref==0.1.2
```

Now, suppose you were setting up a new Python installation or performing a deployment to a new machine. If you wanted to recreate your environment, you could simply type the following:

```
bash % pip install -r requirements.txt
```

This will download, build, and install everything in requirements.txt for you—very nice.

Virtual Environments and Deployments

Once you've mastered the basics of installing packages, you might think that it's the end of the story. After all, how many times are you actually going to sit around installing packages? As it turns out, it might be a lot more often than you think.

One of the more popular extensions to Python is the virtualenv tool (https://pypi.python.org/pypi/virtualenv). virtualenv allows you to make entirely new Python environments for working on new versions of code, experimentation, and testing things out. To make a new virtual environment, simply type the following:

```
bash % virtualenv spam
```

This creates a new directory spam/ in which you will find a new Python installation. This installation is actually a "blank slate" of sorts. The directory spam/bin includes Python as well as easy_install, pip, and virtualenv. No other third-party extensions are included. But that's the whole idea—with a virtual environment you get to start over.

Not to worry! Remember the requirements.py file you just created with pip? Let's recreate our setup in the new virtual environment with a single command:

```
bash % spam/bin/pip install -r requirements.txt
```

This will churn away for a while, but when it's done, you'll have a brand new Python environment with everything installed. Because it's an isolated environment, you can continue to experiment with the setup without breaking the default Python installation on your machine.

Naturally, a similar process can be used if you're deploying a Python application to new machines or to workers out on the cloud. Simply make sure you distribute the requirements.txt file and use it to recreate the environment when you need it.

A Performance Headache

If you've made it this far, you will have made recreating your Python environment easy; it all works fine except for one huge headache, which is the performance of it all. When you type the command `pip install -r requirements.txt`, all of the required packages will be downloaded, compiled, and installed from source. Although there are ways to cache the source locally and avoid the download step, the compilation and installation process can take a substantial amount of time. For example, installing a new environment from the requirements.txt file shown here takes a little more than nine minutes on my machine. Much of that time is spent running the C compiler for the numpy and pandas extensions.

Although that might not seem like much time, it can add up quickly if you find yourself making many virtual environments or recreating the Python environment as part of a deployment script. Surely there should be some way to perform a binary installation from pre-built packages instead. Wouldn't that be much faster? Yes, it would.

Enter Wheels

The newly introduced "wheel" standard is an effort to solve this problem. In a nutshell, a wheel is simply a pre-built Python package. Because it's pre-built, none of the usual source compilation steps are necessary. Instead, all of its contents can simply be copied into place.

To see how wheel works, you first must install the separate wheel package. Just use pip:

```
bash % pip install wheel
```

Next, let's make a special directory for our wheels:

```
bash % mkdir /tmp/wheels
```

Once you're done with that, type the following command using the requirements.txt file from earlier:

```
bash % pip wheel --wheel-dir=/tmp/wheels -r requirements.txt
```

This command will churn away for a while, but when it's done, the /tmp/wheels directory will contain a collection of .whl files like this:

```
bash % ls /tmp/wheels
Django-1.6-py2.py3-none-any.whl
SQLAlchemy-0.8.3-cp27-none-macosx_10_4_x86_64.whl
numpy-1.8.0-cp27-none-macosx_10_4_x86_64.whl
pandas-0.12.0-cp27-none-macosx_10_4_x86_64.whl
ply-3.4-py27-none-any.whl
python_dateutil-2.2-py27-none-any.whl
pytz-2013.8-py27-none-any.whl
requests-2.0.1-py27-none-any.whl
six-1.4.1-py27-none-any.whl
virtualenv-1.10.1-py27-none-any.whl
wsgiref-0.1.2-py27-none-any.whl
```

Okay, that's interesting, but what's the point, you ask? The real benefit comes when you later want to recreate your Python environment. Using the wheels directory as a kind of cache, type the following commands to make a new virtual environment:

```
bash % virtualenv spam2
bash % spam2/bin/pip install --use-wheel --no-index --find-links=/tmp/wheels -r requirements.txt
```

This will completely recreate your Python environment exactly as before; however, instead of taking nine more minutes, it now only takes four seconds. That's a speedup of about 13,500%, in case you were keeping track. Needless to say, this ability to almost instantly recreate your Python environment on a moment's notice is interesting.

Under the Covers

The gory details of what's going on inside a wheel file can be found in PEP 427 (http://www.python.org/dev/peps/pep-0427). To be honest, this document mostly made my head hurt, and it did not fully illuminate the big idea at work. Thus, here are the big picture details that you might care to know. A .whl file is simply a .zip file containing the compiled/built package. Everything needed to make the package work is contained inside. For example:

```
bash % unzip -l ply-3.4-py27-none-any.whl
Archive: ply-3.4-py27-none-any.whl
  Length     Date    Time    Name
 ---------  -----   -----   -----
       82  11-25-13  12:02   ply/__init__.py
    33040  11-25-13  12:02   ply/cpp.py
     3170  11-25-13  12:02   ply/ctokens.py
    40739  11-25-13  12:02   ply/lex.py
   128492  11-25-13  12:02   ply/yacc.py
      518  11-25-13  12:08   ply-3.4.dist-info/DESCRIPTION.rst
      439  11-25-13  12:08   ply-3.4.dist-info/pydist.json
        4  11-25-13  12:08   ply-3.4.dist-info/top_level.txt
       93  11-25-13  12:08   ply-3.4.dist-info/WHEEL
      808  11-25-13  12:08   ply-3.4.dist-info/METADATA
      804  11-25-13  12:08   ply-3.4.dist-info/RECORD
 ---------                    -------
   208189                    11 files
```

When a wheel is installed, the files are moved into place in the normal site-packages directory. None of the usual "build" or "compile" steps take place. Needless to say, this is the reason why installing a wheel is super fast.

The other important detail is that the file name for wheels encodes platform-specific details as appropriate. For example, many packages include some component of C code. For these packages, the name will encode information about the underlying architecture. For example, the wheel created for pandas has the following name on my system:

```
pandas-0.12.0-cp27-none-macosx_10_4_x86_64.whl
```

This becomes useful if you're supporting different kinds of machines (Linux and Mac OS X) or different architectures (32-bit vs. 64-bit). Essentially, you can build a wheel cache with the different versions, and the appropriate version will be used during installation.

Big Picture: Using Wheels in Practice

Currently there is an effort to have the authors of commonly used Python packages upload their code to the Python Package Index as wheels in addition to their normal source distributions. As more users start relying on the wheel format, this will make installation faster; however, I also think that this is the least interesting aspect of wheels. This is because you can still reap their benefits regardless of whether or not a wheel is actually uploaded by a package author.

Imagine that you are the maintainer of Python at your company. Internally, you might have an officially approved version of the interpreter as well as a standard set of third-party libraries that you use in all of your applications. As the Python maintainer, you've probably already written some scripts or instructions for setting up the officially approved Python environment on a new machine. All of this works, but it's also a bit slow.

With wheels, you basically can create your own custom package repository of pre-built extensions. If you point users and installation scripts at that repository, you can turn that laborious installation process into an operation that only takes a few seconds, which is pretty neat. Also there are interesting implications for scripts that push code out to clusters and other large system installations.

Alternatively, imagine that you're the maintainer of an application in your organization and you need to give it out to users. These users may have Python on their machines, but perhaps not all of the compilation tools needed to build complex extensions such as those involve C code. Not to worry—you could create a little install script that points them at your custom wheel repository. When they install your application, they'll get all of the pre-built bits that you've made for them. Again, that can solve a lot of problems.

More Information

Official information about wheels can be found in PEP 427 (http://www.python.org/dev/peps/pep-0427). Tutorials and documentation can be found at http://wheel.readthedocs.org/en/latest/. There, you will find even more advanced material, such as how to attach digital signatures to wheels. Although wheels are new, they have been blessed officially as the Python standard. You're sure to be seeing more of them in the future.