wheel Documentation

Release 0.29.0

Daniel Holth

Jan 03, 2018

Contents

A built-package format for Python.

A wheel is a ZIP-format archive with a specially formatted filename and the .whl extension. It is designed to contain all the files for a PEP 376 compatible install in a way that is very close to the on-disk format. Many packages will be properly installed with only the "Unpack" step (simply extracting the file onto sys.path), and the unpacked archive preserves enough information to "Spread" (copy data and scripts to their final locations) at any later time.

The wheel project provides a *bdist_wheel* command for setuptools (requires setuptools >= 0.8.0). Wheel files can be installed with a newer *pip* from https://github.com/pypa/pip or with wheel's own command line utility.

The wheel documentation is at http://wheel.rtfd.org/. The file format is documented in PEP 427 (http://www.python.org/dev/peps/pep-0427/).

The reference implementation is at https://github.com/pypa/wheel

CHAPTER 1

Why not egg?

Python's egg format predates the packaging related standards we have today, the most important being PEP 376 "Database of Installed Python Distributions" which specifies the .dist-info directory (instead of .egg-info) and PEP 426 "Metadata for Python Software Packages 2.0" which specifies how to express dependencies (instead of requires.txt in .egg-info).

Wheel implements these things. It also provides a richer file naming convention that communicates the Python implementation and ABI as well as simply the language version used in a particular package.

Unlike .egg, wheel will be a fully-documented standard at the binary level that is truly easy to install even if you do not want to use the reference implementation.

CHAPTER 2

Code of Conduct

Everyone interacting in the wheel project's codebases, issue trackers, chat rooms, and mailing lists is expected to follow the PyPA Code of Conduct.

CHAPTER 3

Usage

The current version of wheel can be used to speed up repeated installations by reducing the number of times you have to compile your software. When you are creating a virtualenv for each revision of your software the savings can be dramatic. This example packages pyramid and all its dependencies as wheels, and then installs pyramid from the built packages:

```
# Make sure you have the latest pip that supports wheel
pip install --upgrade pip

# Install wheel
pip install wheel

# Build a directory of wheels for pyramid and all its dependencies
pip wheel --wheel-dir=/tmp/wheelhouse pyramid

# Install from cached wheels
pip install --use-wheel --no-index --find-links=/tmp/wheelhouse pyramid

# Install from cached wheels remotely
pip install --use-wheel --no-index --find-links=https://wheelhouse.example.com/␣
↪pyramid
```

For lxml, an up to 3-minute "search for the newest version and compile" can become a less-than-1 second "unpack from wheel".

As a side effect the wheel directory, "/tmp/wheelhouse" in the example, contains installable copies of the exact versions of your application's dependencies. By installing from those cached wheels you can recreate that environment quickly and with no surprises.

To build an individual wheel, run `python setup.py bdist_wheel`. Note that `bdist_wheel` only works with distribute (`import setuptools`); plain `distutils` does not support pluggable commands like `bdist_wheel`. On the other hand `pip` always runs `setup.py` with setuptools enabled.

Wheel also includes its own installer that can only install wheels (not sdists) from a local file or folder, but has the advantage of working even when distribute or pip has not been installed.

Wheel's builtin utility can be invoked directly from wheel's own wheel:

```
$ python wheel-0.21.0-py2.py3-none-any.whl/wheel -h
usage: wheel [-h]

             {keygen,sign,unsign,verify,unpack,install,install-scripts,convert,help}
             ...

positional arguments:
  {keygen,sign,unsign,verify,unpack,install,install-scripts,convert,help}
                        commands
    keygen              Generate signing key
    sign                Sign wheel
    unsign              Remove RECORD.jws from a wheel by truncating the zip
                        file. RECORD.jws must be at the end of the archive.
                        The zip file must be an ordinary archive, with the
                        compressed files and the directory in the same order,
                        and without any non-zip content after the truncation
                        point.
    verify              Verify a wheel. The signature will be verified for
                        internal consistency ONLY and printed. Wheel's own
                        unpack/install commands verify the manifest against
                        the signature and file contents.
    unpack              Unpack wheel
    install             Install wheels
    install-scripts     Install console_scripts
    convert             Convert egg or wininst to wheel
    help                Show this help

optional arguments:
  -h, --help            show this help message and exit
```

CHAPTER 4

Setuptools scripts handling

Setuptools' popular *console_scripts* and *gui_scripts* entry points can be used to generate platform-specific scripts wrappers. Most usefully these wrappers include *.exe* launchers if they are generated on a Windows machine.

As of 0.23.0, *bdist_wheel* no longer places pre-generated versions of these wrappers into the **.data/scripts/* directory of the archive (non-setuptools scripts are still present, of course).

If the scripts are needed, use a real installer like *pip*. The wheel tool *python -m wheel install-scripts package [package ...]* can also be used at any time to call setuptools to write the appropriate scripts wrappers.

CHAPTER 5

Defining the Python version

The *bdist_wheel* command automatically determines the correct tags to use for the generated wheel. These are based on the Python interpreter used to generate the wheel and whether the project contains C extension code or not. While this is usually correct for C code, it can be too conservative for pure Python code. The bdist_wheel command therefore supports two flags that can be used to specify the Python version tag to use more precisely:

```
--universal        Specifies that a pure-python wheel is "universal"
                   (i.e., it works on any version of Python).  This
                   equates to the tag "py2.py3".
--python-tag XXX   Specifies the precise python version tag to use for
                   a pure-python wheel.
--py-limited-api {cp32|cp33|cp34|...}
                   Specifies Python Py_LIMITED_API compatibility with
                   the version of CPython passed and later versions.
                   The wheel will be tagged cpNN.abi3.{arch} on CPython 3.
                   This flag does not affect Python 2 builds or alternate
                   Python implementations.

                   To conform to the limited API, all your C
                   extensions must use only functions from the limited
                   API, pass Extension(py_limited_api=True) and e.g.
                   #define Py_LIMITED_API=0x03020000 depending on
                   the exact minimun Python you wish to support.
```

The –universal and –python-tag flags have no effect when used on a project that includes C extension code.

The default for a pure Python project (if no explicit flags are given) is “pyN” where N is the major version of the Python interpreter used to build the wheel. This is generally the correct choice, as projects would not typically ship different wheels for different minor versions of Python.

A reasonable use of the *–python-tag* argument would be for a project that uses Python syntax only introduced in a particular Python version. There are no current examples of this, but if wheels had been available when Python 2.5 was released (the first version containing the *with* statement), wheels for a project that used the *with* statement would typically use *–python-tag py25*. However, unless a separate version of the wheel was shipped which avoided the use of the new syntax, there is little benefit in explicitly marking the tag in this manner.

Typically, projects would not specify Python tags on the command line, but would use *setup.cfg* to set them as a project default:

```
[bdist_wheel]
universal=1
```

or:

```
[bdist_wheel]
python-tag = py32
```

CHAPTER 6

Defining conditional dependencies

In wheel, the only way to have conditional dependencies (that might only be needed on certain platforms) is to use environment markers as defined by **PEP 426**.

As of wheel 0.24.0, the recommended way to do this is in the setuptools `extras_require` parameter. A `:` separates the extra name from the marker. Wheel's own setup.py has an example:

```
extras_require={
    ':python_version=="2.6"': ['argparse'],
    'signatures': ['keyring'],
    'signatures:sys_platform!="win32"': ['pyxdg'],
    'faster-signatures': ['ed25519ll'],
    'tool': []
},
```

Leaving out the name of the extra (like with "argparse" here) means that only the conditions after `:` determine whether the dependencies will be installed or not.

As of setuptools 36.2.1, you can pass extras as part of `install_requires`. The above requirements could thus be written like this:

```
install_requires=[
    'argparse; python_version=="2.6"',
    'keyring; extra=="signatures"',
    'pyxdg; extra=="signatures" and sys_platform!="win32"',
    'ed25519ll; extra=="faster-signatures"'
]
```

Alternatively (as of setuptools 36.2.7), you can specify your requirements in the `[options]` section of your setup.cfg:

```
[options]
install_requires =
    argparse; python_version=="2.6"
    keyring; extra=="signatures"
```

```
    pyxdg; extra=="signatures" and sys_platform!="win32"
    ed25519ll; extra=="faster-signatures"
```

Warning: Specifying extras via `install_requires` does not yet work with pip (v9.0.1 as of this writing).

CHAPTER 7

Including the license in the generated wheel file

Several open source licenses require the license text to be included in every distributable artifact of the project. Currently, the only way to to do this with "wheel" is to specify the `license_file` key in the `[metadata]` section of the project's `setup.cfg`:

```
[metadata]
license_file = LICENSE.txt
```

The file path should be relative to the project root. The file will be packaged as `LICENSE.txt` (regardless of the original name) in the `.dist-info` directory in the wheel.

There is currently no way to include multiple license related files, but this is going to change in the near future. You can track the progress by subscribing to issue 138 on Github.

CHAPTER 8

Automatically sign wheel files

Wheel contains an experimental digital signatures scheme based on Ed25519 signatures; these signatures are unrelated to pgp/gpg signatures and do not include a trust model.

python setup.py bdist_wheel will automatically sign wheel files if the environment variable *WHEEL_TOOL* is set to the path of the *wheel* command line tool.:

```
# Install wheel with dependencies for generating signatures
$ pip install wheel[signatures]
# Generate a signing key (only once)
$ wheel keygen

$ export WHEEL_TOOL=/path/to/wheel
$ python setup.py bdist_wheel
```

Signing is done in a subprocess because it is not convenient for the build environment to contain bindings to the keyring and cryptography libraries. The keyring library may not be able to find your keys (choosing a different key storage back end based on available dependencies) unless you run it from the same environment used for keygen.

Note: You can also include the `faster-signatures` extra when installing "wheel" to improve the performance of wheel signing.

CHAPTER 9

Format

The wheel format is documented as PEP 427 “The Wheel Binary Package Format…” (http://www.python.org/dev/peps/pep-0427/).

CHAPTER 10

Slogans

Wheel

- Because 'newegg' was taken.
- Python packaging - reinvented.
- A container for cheese.
- It makes it easier to roll out software.

10.1 The Story of Wheel

I was impressed with Tarek's packaging talk at PyCon 2010, and I admire PEP 345 (Metadata for Python Software Packages 1.2) and PEP 376 (Database of Installed Python Distributions) which standardize a richer metadata format and show how distributions should be installed on disk. So naturally with all the hubbub about *packaging* in Python 3.3, I decided to try it to reap the benefits of a more standardized and predictable Python packaging experience.

I began by converting *cryptacular*, a password hashing package which has a simple C extension, to use setup.cfg. I downloaded the Python 3.3 source, struggled with the difference between setup.py and setup.cfg syntax, fixed the *define_macros* feature, stopped using the missing *extras* functionality, and several hours later I was able to generate my *METADATA* from *setup.cfg*. I rejoiced at my newfound freedom from the tyranny of arbitrary code execution during the build and install process.

It was a lot of work. The package is worse off than before, and it can't be built or installed without patching the Python source code itself.

It was about that time that distutils-sig had a discussion about the need to include a generated setup.cfg from setup.cfg because setup.cfg wasn't static enough. Wait, what?

Of course there is a different way to massively simplify the install process. It's called built or binary packages. You never have to run *setup.py* because there is no *setup.py*. There is only METADATA aka PKG-INFO. Installation has two steps: 'build package'; 'install package', and you can skip the first step, have someone else do it for you, do it on another machine, or install the build system from a binary package and let the build system handle the building. The build is still complicated, but installation is simple.

With the binary package strategy people who want to install use a simple, compatible installer, and people who want to package use whatever is convenient for them for as long as it meets their needs. No one has to rewrite *setup.py* for their own or the 20k+ other packages on PyPi unless a different build system does a better job.

Wheel is my attempt to benefit from the excellent distutils-sig work without having to fix the intractable *distutils* software itself. Like METADATA and .dist-info directories but unlike Extension(), it's simple enough that there really could be alternate implementations; the simplest (but less than ideal) installer is nothing more than "unzip archive.whl" somewhere on sys.path.

If you've made it this far you probably wonder whether I've heard of eggs. Some comparisons:

- Wheel is an installation format; egg is importable. Wheel archives do not need to include .pyc and are less tied to a specific Python version or implementation. Wheel can install (pure Python) packages built with previous versions of Python so you don't always have to wait for the packager to catch up.
- Wheel uses .dist-info directories; egg uses .egg-info. Wheel is compatible with the new world of Python *packaging* and the new concepts it brings.
- Wheel has a richer file naming convention for today's multi-implementation world. A single wheel archive can indicate its compatibility with a number of Python language versions and implementations, ABIs, and system architectures. Historically the ABI has been specific to a CPython release, but when we get a longer-term ABI, wheel will be ready.
- Wheel is lossless. The first wheel implementation *bdist_wheel* always generates *egg-info*, and then converts it to a *.whl*. Later tools will allow for the conversion of existing eggs and bdist_wininst distributions.
- Wheel is versioned. Every wheel file contains the version of the wheel specification and the implementation that packaged it. Hopefully the next migration can simply be to Wheel 2.0.

I hope you will benefit from wheel.

Index

P