PEP 427 -- The Wheel Binary Package Format 1.0

PEP:	427
Title:	The Wheel Binary Package Format 1.0
Author:	Daniel Holth <dholth at gmail.com>
BDFL-Delegate:	Nick Coghlan <ncoghlan at gmail.com>
Discussions-To:	<distutils-sig at python.org (mailto:distutils-sig@python.org?subject=PEP%20427)>
Status:	Final
Type:	Standards Track
Created:	20-Sep-2012
Post-History:	18-Oct-2012, 15-Feb-2013
Resolution:	https://mail.python.org/pipermail/python-dev/2013-February/124103.html (https://mail.python.org/pipermail/python-dev/2013-February/124103.html)

Contents

Canonical specification (#id3)

The canonical version of the wheel format specification is now maintained at https://packaging.python.org/specifications/binary-distribution-format/ (https://packaging.python.org/specifications/binary-distribution-format/) . This may contain amendments relative to this PEP.

Abstract (#id4)

This PEP describes a built-package format for Python called "wheel".

A wheel is a ZIP-format archive with a specially formatted file name and the `.whl` extension. It contains a single distribution nearly as it would be installed according to PEP 376 (/dev/peps/pep-0376) with a particular installation scheme. Although a specialized installer is recommended, a wheel file may be installed by simply unpacking into site-packages with the standard 'unzip' tool while preserving enough information to spread its contents out onto their final paths at any later time.

PEP Acceptance (#id5)

This PEP was accepted, and the defined wheel version updated to 1.0, by Nick Coghlan on 16th February, 2013 [1] (#id2)

Rationale (#id6)

Python needs a package format that is easier to install than sdist. Python's sdist packages are defined by and require the distutils and setuptools build systems, running arbitrary code to build-and-install, and re-compile, code just so it can be installed into a new virtualenv. This system of conflating build-install is slow, hard to maintain, and hinders innovation in both build systems and installers.

Wheel attempts to remedy these problems by providing a simpler interface between the build system and the installer. The wheel binary package format frees installers from having to know about the build system, saves time by amortizing compile time over many installations, and removes the need to install a build system in the target environment.

Details (#id7)

Installing a wheel 'distribution-1.0-py32-none-any.whl' (#id8)

Wheel installation notionally consists of two phases:

- Unpack.
  1. Parse `distribution-1.0.dist-info/WHEEL`.
  2. Check that installer is compatible with Wheel-Version. Warn if minor version is greater, abort if major version is greater.
  3. If Root-Is-Purelib == 'true', unpack archive into purelib (site-packages).
  4. Else unpack archive into platlib (site-packages).
- Spread.
  1. Unpacked archive includes `distribution-1.0.dist-info/` and (if there is data) `distribution-1.0.data/`.
  2. Move each subtree of `distribution-1.0.data/` onto its destination path. Each subdirectory of `distribution-1.0.data/` is a key into a dict of destination directories, such as `distribution-1.0.data/(purelib|platlib|headers|scripts|data)`. The initially supported paths are taken from `distutils.command.install`.
  3. If applicable, update scripts starting with `#!python` to point to the correct interpreter.
  4. Update `distribution-1.0.dist-info/RECORD` with the installed paths.
  5. Remove empty `distribution-1.0.data` directory.
  6. Compile any installed .py to .pyc. (Uninstallers should be smart enough to remove .pyc even if it is not mentioned in RECORD.)

Recommended installer features (#id9)

Rewrite `#!python`.

In wheel, scripts are packaged in `{distribution}-{version}.data/scripts/`. If the first line of a file in `scripts/` starts with exactly `b'#!python'`, rewrite to point to the correct interpreter. Unix installers may need to add the +x bit to these files if the archive was created on Windows.

The `b'#!pythonw'` convention is allowed. `b'#!pythonw'` indicates a GUI script instead of a console script.

Generate script wrappers.

In wheel, scripts packaged on Unix systems will certainly not have accompanying .exe wrappers. Windows installers may want to add them during install.

Recommended archiver features (#id10)

Place `.dist-info` at the end of the archive.

Archivers are encouraged to place the `.dist-info` files physically at the end of the archive. This enables some potentially interesting ZIP tricks including the ability to amend the metadata without rewriting the entire archive.

File Format (#id11)

File name convention (#id12)

The wheel filename is `{distribution}-{version}(-{build tag})?-{python tag}-{abi tag}-{platform tag}.whl`.

distribution

Distribution name, e.g. 'django', 'pyramid'.

version

Distribution version, e.g. 1.0.

build tag

Optional build number. Must start with a digit. Acts as a tie-breaker if two wheel file names are the same in all other respects (i.e. name, version, and other tags). Sort as an empty tuple if unspecified, else sort as a two-item tuple with the first item being the initial digits as an `int`, and the second item being the remainder of the tag as a `str`.

language implementation and version tag

E.g. 'py27', 'py2', 'py3'.

abi tag

E.g. 'cp33m', 'abi3', 'none'.

platform tag

E.g. 'linux_x86_64', 'any'.

For example, `distribution-1.0-1-py27-none-any.whl` is the first build of a package called 'distribution', and is compatible with Python 2.7 (any Python 2.7 implementation), with no ABI (pure Python), on any CPU architecture.

The last three components of the filename before the extension are called "compatibility tags." The compatibility tags express the package's basic interpreter requirements and are detailed in PEP 425 (/dev/peps/pep-0425).

Escaping and Unicode (#id13)

Each component of the filename is escaped by replacing runs of non-alphanumeric characters with an underscore _:

```
re.sub("[^\w\d.]+", "_", distribution, re.UNICODE)
```

The archive filename is Unicode. It will be some time before the tools are updated to support non-ASCII filenames, but they are supported in this specification.

The filenames *inside* the archive are encoded as UTF-8. Although some ZIP clients in common use do not properly display UTF-8 filenames, the encoding is supported by both the ZIP specification and Python's `zipfile`.

File contents (#id14)

The contents of a wheel file, where {distribution} is replaced with the name of the package, e.g. `beaglevote` and {version} is replaced with its version, e.g. `1.0.0`, consist of:

1. /, the root of the archive, contains all files to be installed in `purelib` or `platlib` as specified in `WHEEL`. `purelib` and `platlib` are usually both `site-packages`.

2. `{distribution}-{version}.dist-info/` contains metadata.

3. `{distribution}-{version}.data/` contains one subdirectory for each non-empty install scheme key not already covered, where the subdirectory name is an index into a dictionary of install paths (e.g. `data`, `scripts`, `include`, `purelib`, `platlib`).

4. Python scripts must appear in `scripts` and begin with exactly `b'#!python'` in order to enjoy script wrapper generation and `#!python` rewriting at install time. They may have any or no extension.

5. `{distribution}-{version}.dist-info/METADATA` is Metadata version 1.1 or greater format metadata.

6. `{distribution}-{version}.dist-info/WHEEL` is metadata about the archive itself in the same basic key: value format:

```
Wheel-Version: 1.0
Generator: bdist_wheel 1.0
Root-Is-Purelib: true
Tag: py2-none-any
Tag: py3-none-any
Build: 1
```

7. `Wheel-Version` is the version number of the Wheel specification.

8. `Generator` is the name and optionally the version of the software that produced the archive.

9. `Root-Is-Purelib` is true if the top level directory of the archive should be installed into purelib; otherwise the root should be installed into platlib.

10. `Tag` is the wheel's expanded compatibility tags; in the example the filename would contain `py2.py3-none-any`.

11. `Build` is the build number and is omitted if there is no build number.

12. A wheel installer should warn if Wheel-Version is greater than the version it supports, and must fail if Wheel-Version has a greater major version than the version it supports.

13. Wheel, being an installation format that is intended to work across multiple versions of Python, does not generally include .pyc files.

14. Wheel does not contain setup.py or setup.cfg.

This version of the wheel specification is based on the distutils install schemes and does not define how to install files to other locations. The layout offers a superset of the functionality provided by the existing wininst and egg binary formats.

The .dist-info directory (#id15)

1. Wheel .dist-info directories include at a minimum METADATA, WHEEL, and RECORD.
2. METADATA is the package metadata, the same format as PKG-INFO as found at the root of sdists.
3. WHEEL is the wheel metadata specific to a build of the package.
4. RECORD is a list of (almost) all the files in the wheel and their secure hashes. Unlike PEP 376 (/dev/peps/pep-0376), every file except RECORD, which cannot contain a hash of itself, must include its hash. The hash algorithm must be sha256 or better; specifically, md5 and sha1 are not permitted, as signed wheel files rely on the strong hashes in RECORD to validate the integrity of the archive.
5. PEP 376 (/dev/peps/pep-0376)'s INSTALLER and REQUESTED are not included in the archive.
6. RECORD.jws is used for digital signatures. It is not mentioned in RECORD.
7. RECORD.p7s is allowed as a courtesy to anyone who would prefer to use S/MIME signatures to secure their wheel files. It is not mentioned in RECORD.
8. During extraction, wheel installers verify all the hashes in RECORD against the file contents. Apart from RECORD and its signatures, installation will fail if any file in the archive is not both mentioned and correctly hashed in RECORD.

The .data directory (#id16)

Any file that is not normally installed inside site-packages goes into the .data directory, named as the .dist-info directory but with the .data/ extension:

```
distribution-1.0.dist-info/

distribution-1.0.data/
```

The .data directory contains subdirectories with the scripts, headers, documentation and so forth from the distribution. During installation the contents of these subdirectories are moved onto their destination paths.

Signed wheel files (#id17)

Wheel files include an extended RECORD that enables digital signatures. PEP 376 (/dev/peps/pep-0376)'s RECORD is altered to include a secure hash `digestname=urlsafe_b64encode_nopad(digest)` (urlsafe base64 encoding with no trailing = characters) as the second column instead of an md5sum. All possible entries are hashed, including any generated files such as .pyc files, but not RECORD which cannot contain its own hash. For example:

```
file.py,sha256=AVTFPZpEKzuHr7OvQZmhaU3LvwKz06AJw8mT\_pNh2yI,3144
distribution-1.0.dist-info/RECORD,,
```

The signature file(s) RECORD.jws and RECORD.p7s are not mentioned in RECORD at all since they can only be added after RECORD is generated. Every other file in the archive must have a correct hash in RECORD or the installation will fail.

If JSON web signatures are used, one or more JSON Web Signature JSON Serialization (JWS-JS) signatures is stored in a file RECORD.jws adjacent to RECORD. JWS is used to sign RECORD by including the SHA-256 hash of RECORD as the signature's JSON payload:

```
{ "hash": "sha256=ADD-r2urObZHcxBW3Cr-vDCu5RJwT4CaRTHiFmbcIYY" }
```

(The hash value is the same format used in RECORD.)

If RECORD.p7s is used, it must contain a detached S/MIME format signature of RECORD.

A wheel installer is not required to understand digital signatures but MUST verify the hashes in RECORD against the extracted file contents. When the installer checks file hashes against RECORD, a separate signature checker only needs to establish that RECORD matches the signature.

See

- RFC 7515 (http://tools.ietf.org/html/rfc7515.html)
- https://datatracker.ietf.org/doc/html/draft-jones-jose-jws-json-serialization.html (https://datatracker.ietf.org/doc/html/draft-jones-jose-jws-json-serialization.html)
- RFC 7517 (http://tools.ietf.org/html/rfc7517.html)
- https://datatracker.ietf.org/doc/html/draft-jones-jose-json-private-key.html (https://datatracker.ietf.org/doc/html/draft-jones-jose-json-private-key.html)

Comparison to .egg (#id18)

1. 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.
2. 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.
3. 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, wheel is ready for the stable ABI.
4. Wheel is lossless. The first wheel implementation bdist_wheel always generates egg-info, and then converts it to a .whl. It is also possible to convert existing eggs and bdist_wininst distributions.
5. 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.
6. Wheel is a reference to the other Python.

FAQ (#id19)

Wheel defines a .data directory. Should I put all my data there? (#id20)

This specification does not have an opinion on how you should organize your code. The .data directory is just a place for any files that are not normally installed inside `site-packages` or on the PYTHONPATH. In other words, you may continue to use `pkgutil.get_data(package, resource)` even though those files will usually not be distributed in wheel's `.data` directory.

Why does wheel include attached signatures? (#id21)

Attached signatures are more convenient than detached signatures because they travel with the archive. Since only the individual files are signed, the archive can be recompressed without invalidating the signature or individual files can be verified without having to download the whole archive.

Why does wheel allow JWS signatures? (#id22)

The JOSE specifications of which JWS is a part are designed to be easy to implement, a feature that is also one of wheel's primary design goals. JWS yields a useful, concise pure-Python implementation.

Why does wheel also allow S/MIME signatures? (#id23)

S/MIME signatures are allowed for users who need or want to use existing public key infrastructure with wheel.

Signed packages are only a basic building block in a secure package update system. Wheel only provides the building block.

What's the deal with "purelib" vs. "platlib"? (#id24)

Wheel preserves the "purelib" vs. "platlib" distinction, which is significant on some platforms. For example, Fedora installs pure Python packages to '/usr/lib/pythonX.Y/site-packages' and platform dependent packages to '/usr/lib64/pythonX.Y/site-packages'.

A wheel with "Root-Is-Purelib: false" with all its files in `{name}-{version}.data/purelib` is equivalent to a wheel with "Root-Is-Purelib: true" with those same files in the root, and it is legal to have files in both the "purelib" and "platlib" categories.

In practice a wheel should have only one of "purelib" or "platlib" depending on whether it is pure Python or not and those files should be at the root with the appropriate setting given for "Root-is-purelib".

Is it possible to import Python code directly from a wheel file? (#id25)

Technically, due to the combination of supporting installation via simple extraction and using an archive format that is compatible with `zipimport`, a subset of wheel files do support being placed directly on `sys.path`. However, while this behaviour is a natural consequence of the format design, actually relying on it is generally discouraged.

Firstly, wheel is designed primarily as a distribution format, so skipping the installation step also means deliberately avoiding any reliance on features that assume full installation (such as being able to use standard tools like `pip` and `virtualenv` to capture and manage dependencies in a way that can be properly tracked for auditing and security update purposes, or integrating fully with the standard build machinery for C extensions by publishing header files in the appropriate place).

Secondly, while some Python software is written to support running directly from a zip archive, it is still common for code to be written assuming it has been fully installed. When that assumption is broken by trying to run the software from a zip archive, the failures can often be obscure and hard to diagnose (especially when they occur in third party libraries). The two most common sources of problems with this are the fact that importing C extensions from a zip archive is not supported by CPython (since doing so is not supported directly by the dynamic loading machinery on any platform) and that when running from a zip archive the `__file__` attribute no longer refers to an ordinary filesystem path, but to a combination path that includes both the location of the zip archive on the filesystem and the relative path to the module inside the archive. Even when software correctly uses the abstract resource APIs internally, interfacing with external components may still require the availability of an actual on-disk file.

Like metaclasses, monkeypatching and metapath importers, if you're not already sure you need to take advantage of this feature, you almost certainly don't need it. If you do decide to use it anyway, be aware that many projects will require a failure to be reproduced with a fully installed package before accepting it as a genuine bug.

References (#id26)

[1] (#id1) PEP acceptance (https://mail.python.org/pipermail/python-dev/2013-February/124103.html (https://mail.python.org/pipermail/python-dev/2013-February/124103.html))

Appendix (#id27)

Example urlsafe-base64-nopad implementation:

```
# urlsafe-base64-nopad for Python 3
import base64

def urlsafe_b64encode_nopad(data):
    return base64.urlsafe_b64encode(data).rstrip(b'=')

def urlsafe_b64decode_nopad(data):
    pad = b'=' * (4 - (len(data) & 3))
    return base64.urlsafe_b64decode(data + pad)
```

Copyright (#id28)

Source: https://github.com/python/peps/blob/master/pep-0427.txt (https://github.com/python/peps/blob/master/pep-0427.txt)