What Are Python Wheels and Why Should You Care?

by Brad Solomon 12 Comments

intermediate python

Table of Contents

Python `.whl` files, or wheels, are a little-discussed part of Python, but they've been a boon to the installation process for Python packages. If you've installed a Python package using `pip`, then chances are that a wheel has made the installation faster and more efficient.

Wheels are a component of the Python ecosystem that helps to make package installs *just work*. They allow for faster installations and more stability in the package distribution process. In this tutorial, you'll dive into what wheels are, what good they serve, and how they've gained traction and made Python even more of a joy to work with.

In this tutorial, you'll learn:

- What wheels are and how they compare to **source distributions**
- How you can use wheels to control the **package installation** process
- How to **create and distribute** wheels for your own Python packages

You'll see examples using popular open source Python packages from both the user's and the developer's perspective.

Setup

To follow along, activate a virtual environment and make sure you have the latest versions of `pip`, `wheel`, and `setuptools` installed:

Shell

```
$ python -m venv env && source ./env/bin/activate
$ python -m pip install -U pip wheel setuptools
Successfully installed pip 20.1 setuptools-46.1.3 wheel-0.34.2
```

That’s all you need to experiment with installing and building wheels!

Python Packaging Made Better: An Intro to Python Wheels

Before you learn how to package a project into a wheel, it helps to know what using one looks like from the user’s side. It may sound backward, but a good way to learn how wheels work is to start by installing something that *isn’t* a wheel.

You can start this experiment by installing a Python package into your environment just as you might normally do. In this case, install uWSGI version 2.0.x:

Shell

```
 1  $ python -m pip install 'uwsgi==2.0.*'
 2  Collecting uwsgi==2.0.*
 3    Downloading uwsgi-2.0.18.tar.gz (801 kB)
 4       |████████████████████████████████| 801 kB 1.1 MB/s
 5  Building wheels for collected packages: uwsgi
 6    Building wheel for uwsgi (setup.py) ... done
 7    Created wheel for uwsgi ... uWSGI-2.0.18-cp38-cp38-macosx_10_15_x86_64.whl
 8    Stored in directory: /private/var/folders/jc/8_hqsz0x1tdbp05 ...
 9  Successfully built uwsgi
10  Installing collected packages: uwsgi
11  Successfully installed uwsgi-2.0.18
```

To fully install uWSGI, `pip` progresses through several distinct steps:

1. On **line 3**, it downloads a TAR file (tarball) named `uwsgi-2.0.18.tar.gz` that’s been compressed with gzip.
2. On **line 6**, it takes the tarball and builds a `.whl` file through a call to `setup.py`.
3. On **line 7**, it labels the wheel `uWSGI-2.0.18-cp38-cp38-macosx_10_15_x86_64.whl`.
4. On **line 10**, it installs the actual package after having built the wheel.

The `tar.gz` tarball that `pip` retrieves is a **source distribution**, or `sdist`, rather than a wheel. In some ways, a `sdist` is the opposite of a wheel.

Note: If you see an error with the uWSGI installation, you may need to install the Python development headers.

A source distribution contains source code. That includes not only Python code but also the source code of any extension modules (usually in C or C++) bundled with the package. With source distributions, extension modules are compiled on the user’s side rather than the developer’s.

Source distributions also contain a bundle of metadata sitting in a directory called `<package-name>.egg-info`. This metadata helps with building and installing the package, but user’s don’t really need to do anything with it.

From the developer’s perspective, a source distribution is what gets created when you run the following command:

Shell

```
$ python setup.py sdist
```

Now try installing a different package, chardet:

Shell

```
$ python -m pip install 'chardet==3.*'
Collecting chardet
  Downloading chardet-3.0.4-py2.py3-none-any.whl (133 kB)
     |████████████████████████████████| 133 kB 1.5 MB/s
Installing collected packages: chardet
Successfully installed chardet-3.0.4
```

You can see a noticeably different output than the uWSGI install.

Installing chardet downloads a `.whl` file directly from PyPI. The wheel name `chardet-3.0.4-py2.py3-none-any.whl` follows a specific naming convention that you'll see later. What's more important from the user's perspective is that there's no build stage when `pip` finds a compatible wheel on PyPI.

From the developer's side, a wheel is the result of running the following command:

Shell

```
$ python setup.py bdist_wheel
```

Why does uWSGI hand you a source distribution while chardet provides a wheel? You can see the reason for this by taking a look at each project's page on PyPI and navigating to the *Download files* area. This section will show you what `pip` actually sees on the PyPI index server:

- **uWSGI** provides only a source distribution (`uwsgi-2.0.18.tar.gz`) for reasons related to the complexity of the project.
- **chardet** provides both a wheel and a source distribution, but `pip` will prefer the wheel *if* it's compatible with your system. You'll see how that compatibility is determined later on.

Another example of the compatibility check used for wheel installation is `psycopg2`, which provides a wide set of wheels for Windows but doesn't provide any for Linux or macOS clients. This means that `pip install psycopg2` could fetch a wheel or a source distribution depending on your specific setup.

To avoid these types of compatibility issues, some packages offer multiple wheels, with each wheel geared toward a specific Python implementation and underlying operating system.

So far, you've seen some of the visible distinctions between a wheel and `sdist`, but what matters more is the impact those differences have on the installation process.

Wheels Make Things Go Fast

Above, you saw a comparison of an installation that fetches a prebuilt wheel and one that downloads a `sdist`. Wheels make the end-to-end installation of Python packages faster for two reasons:

1. All else being equal, wheels are typically **smaller in size** than source distributions, meaning they can move faster across a network.
2. Installing from wheels directly avoids the intermediate step of **building** packages off of the source distribution.

It's almost guaranteed that the chardet install occurred in a fraction of the time required for uWSGI. However, that's arguably an unfair apples-to-oranges comparison since chardet is a significantly smaller and less complex package. With a different command, you can create a more direct comparison that will demonstrate just how much of a difference wheels make.

You can make `pip` ignore its inclination towards wheels by passing the `--no-binary` option:

Shell

```
$ time python -m pip install \
      --no-cache-dir \
      --force-reinstall \
      --no-binary=:all: \
      cryptography
```

This command times the installation of the `cryptography` package, telling `pip` to use a source distribution even if a suitable wheel is available. Including `:all:` makes the rule apply to `cryptography` and all of its dependencies.

On my machine, this takes around *thirty-two seconds* from start to finish. Not only does the install take a long time, but building `cryptography` also requires that you have the OpenSSL development headers present and available to Python.

Note: With `--no-binary`, you may very well see an error about missing header files required for the `cryptography` install, which is part of what can make using source distributions frustrating. If so, the installation section of the `cryptography` docs advises on which libraries and header files you'll need for a particular operating system.

Now you can reinstall `cryptography`, but this time make sure that `pip` uses wheels from PyPI. Because `pip` will prefer a wheel, this is similar to just calling `pip install` with no arguments at all. But in this case, you can make the intent explicit by requiring a wheel with `--only-binary`:

Shell

```
$ time python -m pip install \
      --no-cache-dir \
      --force-reinstall \
      --only-binary=cryptography \
      cryptography
```

This option takes just over four seconds, or *one-eighth* the time that it took when using only source distributions for `cryptography` and its dependencies.

What Is a Python Wheel?

A Python `.whl` file is essentially a ZIP (`.zip`) archive with a specially crafted filename that tells installers what Python versions and platforms the wheel will support.

A wheel is a type of **built distribution**. In this case, *built* means that the wheel comes in a ready-to-install format and allows you to skip the build stage required with source distributions.

Note: It's worth mentioning that despite the use of the term *built*, a wheel doesn't contain `.pyc` files, or compiled Python bytecode.

A wheel filename is broken down into parts separated by hyphens:

Text

```
{dist}-{version}(-{build})?-{python}-{abi}-{platform}.whl
```

Each section in `{brackets}` is a **tag**, or a component of the wheel name that carries some meaning about what the wheel contains and where the wheel will or will not work.

Here's an illustrative example using a `cryptography` wheel:

Text

```
cryptography-2.9.2-cp35-abi3-macosx_10_9_x86_64.whl
```

`cryptography` distributes multiple wheels. Each wheel is a **platform wheel**, meaning it supports only specific combinations of Python versions, Python ABIs, operating systems, and machine architectures. You can break down the naming convention into parts:

- **`cryptography`** is the package name.
- **`2.9.2`** is the package version of `cryptography`. A version is a PEP 440-compliant string such as `2.9.2`, `3.4`, or `3.9.0.a3`.
- **`cp35`** is the Python tag and denotes the Python implementation and version that the wheel demands. The `cp` stands for CPython, the reference implementation of Python, while the `35` denotes Python 3.5. This wheel wouldn't be compatible with Jython, for instance.
- **`abi3`** is the ABI tag. ABI stands for application binary interface. You don't really need to worry about what it entails, but `abi3` is a separate version for the binary compatibility of the Python C API.
- **`macosx_10_9_x86_64`** is the platform tag, which happens to be quite a mouthful. In this case it can be broken down further into sub-parts:
    - **`macosx`** is the macOS operating system.
    - **`10_9`** is the macOS developer tools SDK version used to compile the Python that in turn built this wheel.
    - **`x86_64`** is a reference to x86-64 instruction set architecture.

The final component isn't technically a tag but rather the standard `.whl` file extension. Combined, the above components indicate the target machine that this `cryptography` wheel is designed for.

Now let's turn to a different example. Here's what you saw in the above case for chardet:

Text
```
chardet-3.0.4-py2.py3-none-any.whl
```

You can break this down into its tags:

- **`chardet`** is the package name.
- **`3.0.4`** is the package version of chardet.
- **`py2.py3`** is the Python tag, meaning the wheel supports Python 2 and 3 with any Python implementation.
- **`none`** is the ABI tag, meaning the ABI isn't a factor.
- **`any`** is the platform. This wheel will work on virtually any platform.

The `py2.py3-none-any.whl` segment of the wheel name is common. This is a **universal wheel** that will install with Python 2 or 3 on any platform with any ABI. If the wheel ends in `none-any.whl`, then it's very likely a pure-Python package that doesn't care about a specific Python ABI or CPU architecture.

Another example is the `jinja2` templating engine. If you navigate to the downloads page for the Jinja 3.x alpha release, then you'll see the following wheel:

Text
```
Jinja2-3.0.0a1-py3-none-any.whl
```

Notice the lack of `py2` here. This is a pure-Python project that will work on any Python 3.x version, but it's not a universal wheel because it doesn't support Python 2. Instead, it's called a **pure-Python wheel**.

Note: In 2020, a number of projects are also dropping support for Python 2, which reached end-of-life (EOL) on January 1, 2020. Jinja version 3.x dropped Python 2 support in February 2020.

Here are a few more examples of `.whl` names distributed for some popular open source packages:

Wheel	What It Is
`PyYAML-5.3.1-cp38-cp38-win_amd64.whl`	PyYAML for CPython 3.8 on Windows with AMD64 (x86-64) architecture
`numpy-1.18.4-cp38-cp38-win32.whl`	NumPy for CPython 3.8 on Windows 32-bit
`scipy-1.4.1-cp36-cp36m-macosx_10_6_intel.whl`	SciPy for CPython 3.6 on macOS 10.6 SDK with fat binary (multiple instruction sets)

Now that you have a thorough understanding of what wheels are, it's time to talk about what good they serve.

Advantages of Python Wheels

Here's a testament to wheels from the Python Packaging Authority (PyPA):

> Not all developers have the right tools or experiences to build these components written in these compiled languages, so Python created the wheel, a package format designed to ship libraries with compiled artifacts. In fact, Python's package installer, `pip`, always prefers wheels because installation is always faster, so even pure-Python packages work better with wheels. (Source)

A fuller description is that wheels benefit both users and maintainers of Python packages alike in a handful of ways:

- **Wheels install faster** than source distributions for both pure-Python packages and extension modules.
- **Wheels are smaller** than source distributions. For example, the `six` wheel is about one-third the size of the corresponding source distribution. This differential becomes even more important when you consider that a `pip install` for a single package may actually kick off downloading a chain of dependencies.
- **Wheels cut `setup.py` execution out of the equation.** Installing from a source distribution runs *whatever* is contained in that project's `setup.py`. As pointed out by PEP 427, this amounts to arbitrary code execution. Wheels avoid this altogether.
- **There's no need for a compiler** to install wheels that contain compiled extension modules. The extension module comes included with the wheel targeting a specific platform and Python version.
- **`pip` automatically generates `.pyc` files** in the wheel that match the right Python interpreter.
- **Wheels provide consistency** by cutting many of the variables involved in installing a package out of the equation.

You can use a project's *Download files* tab on PyPI to view the different distributions that are available. For example, pandas distributes a wide array of wheels.

Telling `pip` What to Download

It's possible to exert fine-grained control over `pip` and tell it which format to prefer or avoid. You can use the `--only-binary` and `--no-binary` options to do this. You saw these used in an earlier section on installing the `cryptography` package, but it's worth taking a closer look at what they do:

Shell

```shell
$ pushd "$(mktemp -d)"
$ python -m pip download --only-binary :all: --dest . --no-cache six
Collecting six
  Downloading six-1.14.0-py2.py3-none-any.whl (10 kB)
  Saved ./six-1.14.0-py2.py3-none-any.whl
Successfully downloaded six
```

In this example, you change to a temporary directory to store the download with `pushd "$(mktemp -d)"`. You use `pip download` rather than `pip install` so that you can inspect the resulting wheel, but you can replace `download` with `install` while keeping the same set of options.

You download the `six` module with several flags:

- **`--only-binary :all:`** tells `pip` to constrain itself to using wheels and ignore source distributions. Without this option, `pip` will only *prefer* wheels but will fall back to source distributions in some scenarios.
- **`--dest .`** tells `pip` to download `six` to the current directory.
- **`--no-cache`** tells `pip` not to look in its local download cache. You use this option just to illustrate a live download from PyPI since it's likely you do have a `six` cache somewhere.

I mentioned earlier that a wheel file is essentially a `.zip` archive. You can take this statement literally and treat wheels as such. For instance, if you want to view a wheel's contents, you can use `unzip`:

Shell

```
$ unzip -l six*.whl
Archive:  six-1.14.0-py2.py3-none-any.whl
  Length      Date    Time    Name
---------  ---------- -----   ----
    34074  01-15-2020 18:10   six.py
     1066  01-15-2020 18:10   six-1.14.0.dist-info/LICENSE
     1795  01-15-2020 18:10   six-1.14.0.dist-info/METADATA
      110  01-15-2020 18:10   six-1.14.0.dist-info/WHEEL
        4  01-15-2020 18:10   six-1.14.0.dist-info/top_level.txt
      435  01-15-2020 18:10   six-1.14.0.dist-info/RECORD
---------                     -------
    37484                     6 files
```

`six` is a special case: it's actually a single Python module rather than a complete package. Wheel files can also be significantly more complex, as you'll see later on.

In contrast to `--only-binary`, you can use `--no-binary` to do the opposite:

Shell

```
$ python -m pip download --no-binary :all: --dest . --no-cache six
Collecting six
  Downloading six-1.14.0.tar.gz (33 kB)
  Saved ./six-1.14.0.tar.gz
Successfully downloaded six
$ popd
```

The only change in this example is the switch to `--no-binary :all:`. This tells `pip` to ignore wheels even if they're available and instead download a source distribution.

When might `--no-binary` be useful? Here are a few cases:

- **The corresponding wheel is broken.** This is an irony of wheels. They're designed to make things break less often, but in some cases a wheel can be misconfigured. In this case, downloading and building the source distribution for yourself may be a working alternative.

- **You want to apply a small change or patch file** to the project and then install it. This is an alternative to cloning the project from its version control system URL.

You can also use the flags described above with `pip install`. Additionally, instead of `:all:`, which will apply the `--only-binary` rule not just to the package you're installing but to all of its dependencies, you can pass `--only-binary` and `--no-binary` a list of specific packages to apply that rule to.

Here are a few examples for installing the URL library `yarl`. It contains Cython code and depends on `multidict`, which contains pure C code. There are several options to strictly use or strictly ignore wheels for `yarl` and its dependencies:

Shell

```
$ # Install `yarl` and use only wheels for yarl and all dependencies
$ python -m pip install --only-binary :all: yarl

$ # Install `yarl` and use wheels only for the `multidict` dependency
$ python -m pip install --only-binary multidict yarl

$ # Install `yarl` and don't use wheels for yarl or any dependencies
$ python -m pip install --no-binary :all: yarl

$ # Install `yarl` and don't use wheels for the `multidict` dependency
$ python -m pip install --no-binary multidict yarl
```

In this section, you got a glimpse of how to fine-tune the distribution types that `pip install` will use. While a regular `pip install` should work with no options, it's helpful to know these options for special cases.

The `manylinux` Wheel Tag

Linux comes in many variants and flavors, such as Debian, CentOS, Fedora, and Pacman. Each of these may use slight variations in shared libraries, such as `libncurses`, and core C libraries, such as `glibc`.

If you're writing a C/C++ extension, then this could create a problem. A source file written in C and compiled on Ubuntu Linux isn't guaranteed to be executable on a CentOS machine or an Arch Linux distribution. Do you need to build a separate wheel for each and every Linux variant?

Luckily, the answer is no, thanks to a specially designed set of tags called the **`manylinux`** platform tag family. There are currently three variations:

1. **`manylinux1`** is the original format specified in PEP 513.
2. **`manylinux2010`** is an update specified in PEP 571 that upgrades to CentOS 6 as the underlying OS on which the Docker images are based. The rationale is that CentOS 5.11, which is where the list of allowed libraries in `manylinux1` comes from, reached EOL in March 2017 and stopped receiving security patches and bug fixes.
3. **`manylinux2014`** is an update specified in PEP 599 that upgrades to CentOS 7 since CentOS 6 is scheduled to reach EOL in November 2020.

You can find an example of `manylinux` distributions within the pandas project. Here are two (out of many) from the list of available pandas downloads from PyPI:

Text

```
pandas-1.0.3-cp37-cp37m-manylinux1_x86_64.whl
pandas-1.0.3-cp37-cp37m-manylinux1_i686.whl
```

In this case, pandas has built `manylinux1` wheels for CPython 3.7 supporting both x86-64 and i686 architectures.

At its core, `manylinux` is a Docker image built off a certain version of the CentOS operating system. It comes bundled with a compiler suite, multiple versions of Python and `pip`, and an allowed set of shared libraries.

Note: The term **allowed** indicates a low-level library that is assumed to be present by default on almost all Linux systems. The idea is that the dependency should exist on the base operating system without the need for an additional install.

As of mid-2020, `manylinux1` is still the predominant `manylinux` tag. One reason for this might just be habit. Another might be that support on the client (user) side for `manylinux2010` and above is limited to more recent versions of `pip`:

Tag	Requirement
`manylinux1`	`pip` 8.1.0 or later

Tag	Requirement
`manylinux2010`	`pip` 19.0 or later
`manylinux2014`	`pip` 19.3 or later

In other words, if you're a package developer building `manylinux2010` wheels, then someone using your package will need `pip` 19.0 (released in January 2019) or later to let `pip` find and install `manylinux2010` wheels from PyPI.

Luckily, virtual environments have become more common, meaning that developers can update a virtual environment's `pip` without touching the system `pip`. This isn't always the case, however, and some Linux distributions still ship with outdated versions of `pip`.

All that is to say, if you're installing Python packages on a Linux host, then consider yourself fortunate if the package maintainer has gone out of their way to create `manylinux` wheels. This will almost guarantee a hassle-free installation of the package regardless of your specific Linux variant or version.

Caution: Be advised that PyPI wheels don't work on Alpine Linux (or BusyBox). This is because Alpine uses `musl` in place of the standard `glibc`. The `musl libc` library bills itself as "a new `libc` striving to be fast, simple, lightweight, free, and correct." Unfortunately, when it comes to wheels, `glibc` it is not.

Security Considerations With Platform Wheels

One feature of wheels worth considering from a user security standpoint is that wheels are potentially subject to version rot because they bundle a binary dependency rather than allowing that dependency to be updated by your system package manager.

For example, if a wheel incorporates the `libfortran` shared library, then distributions of that wheel will use the `libfortran` version that they were bundled with even if you upgrade your own machine's version of `libfortran` with a package manager such as `apt`, `yum`, or `brew`.

If you're developing in an environment with heightened security precautions, this feature of some platform wheels is something to be mindful of.

Calling All Developers: Build Your Wheels

The title of this tutorial asks, "Why Should You Care?" As a developer, if you plan to distribute a Python package to the community, then you should care immensely about distributing wheels for your project because they make the installation process cleaner and less complex for end users.

The more target platforms that you can support with compatible wheels, the fewer GitHub issues you'll see titled something like "Installation broken on Platform XYZ." Distributing wheels for your Python package makes it objectively less likely that users of the package will encounter issues during installation.

The first thing you need to do to build a wheel locally is to install `wheel`. It doesn't hurt to make sure that `setuptools` is up to date, too:

Shell

```
$ python -m pip install -U wheel setuptools
```

The next few sections will walk you through building wheels for a variety of different scenarios.

Different Types of Wheels

As touched on throughout this tutorial, there are several different variations of wheels, and the wheel's type is reflected in its filename:

- A **universal wheel** contains `py2.py3-none-any.whl`. It supports both Python 2 and Python 3 on any OS and platform. The majority of wheels listed on the Python Wheels website are universal wheels.
- A **pure-Python wheel** contains either `py3-none-any.whl` or `py2.none-any.whl`. It supports either Python 3 or Python 2, but not both. It's otherwise the same as a universal wheel, but it'll be labeled with either `py2` or `py3` rather than the `py2.py3` label.
- A **platform wheel** supports a specific Python version and platform. It contains segments indicating a specific Python version, ABI, operating system, or architecture.

The differences between wheel types are determined by which version(s) of Python they support and whether they target a specific platform. Here's a condensed summary of the differences between wheel variations:

Wheel Type	Supports Python 2 and 3	Supports Every ABI, OS, and Platform
Universal	✓	✓
Pure-Python		✓
Platform		

As you'll see next, you can build universal wheels and pure-Python wheels with relatively little setup, but platform wheels may require a few additional steps.

Building a Pure-Python Wheel

You can build a pure-Python wheel or a universal wheel for any project using `setuptools` with just a single command:

Shell

```
$ python setup.py sdist bdist_wheel
```

This will create both a source distribution (`sdist`) and a wheel (`bdist_wheel`). By default, both will be placed in `dist/` under the current directory. To see for yourself, you can build a wheel for HTTPie, a command-line HTTP client written in Python, alongside a `sdist`.

Here's the result of building both types of distributions for the HTTPie package:

Shell

```
$ git clone -q git@github.com:jakubroztocil/httpie.git
$ cd httpie
$ python setup.py -q sdist bdist_wheel
$ ls -1 dist/
httpie-2.2.0.dev0-py3-none-any.whl
httpie-2.2.0.dev0.tar.gz
```

That's all it takes. You clone the project, move into its root directory, and then call `python setup.py sdist bdist_wheel`. You can see that `dist/` contains both a wheel and a source distribution.

The resulting distributions get put in `dist/` by default, but you can change that with the `-d`/`--dist-dir` option. You could put them in a temporary directory instead for build isolation:

Shell

```
$ tempdir="$(mktemp -d)"  # Create a temporary directory
$ file "$tempdir"
/var/folders/jc/8_kd8uusys7ak09_lpmn30rw0000gk/T/tmp.GIXy7XKV: directory

$ python setup.py sdist -d "$tempdir"
$ python setup.py bdist_wheel --dist-dir "$tempdir"
$ ls -1 "$tempdir"
httpie-2.2.0.dev0-py3-none-any.whl
httpie-2.2.0.dev0.tar.gz
```

You can combine the `sdist` and `bdist_wheel` steps into one because `setup.py` can take multiple subcommands:

Shell

```
$ python setup.py sdist -d "$tempdir" bdist_wheel -d "$tempdir"
```

As shown here, you'll need to pass options such as `-d` to each subcommand.

Specifying a Universal Wheel

A universal wheel is a wheel for a pure-Python project that supports both Python 2 and 3. There are multiple ways to tell `setuptools` and `distutils` that a wheel should be universal.

Option 1 is to specify the option in your project's [`setup.cfg`] file:

Config File

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

Option 2 is to pass the aptly named `--universal` flag at the command line:

Shell

```
$ python setup.py bdist_wheel --universal
```

Option 3 is to tell `setup()` itself about the flag using its `options` parameter:

Python

```
# setup.py
from setuptools import setup

setup(
    # ....
    options={"bdist_wheel": {"universal": True}}
    # ....
)
```

While any of these three options should work, the first two are used most frequently. You can see an example of this in the chardet setup configuration. After that, you can use the `bdist_wheel` command as shown previously:

Shell

```
$ python setup.py sdist bdist_wheel
```

The resulting wheel will be equivalent no matter which option you choose. The choice largely comes down to developer preference and which workflow is best for you.

Building a Platform Wheel (macOS and Windows)

Binary distributions are a subset of **built distributions** that contain compiled extensions. **Extensions** are non-Python dependencies or components of your Python package.

Usually, that means your package contains an extension module or depends on a library written in a statically typed language such as C, C++, Fortran, or even Rust or Go. **Platform wheels** exist to target individual platforms primarily because they contain or depend on extension modules.

With all that said, it's high time to build a platform wheel!

Depending on your existing development environment, you may need to go through an additional prerequisite step or two to build platform wheels. The steps below will help you to get set up for building C and C++ extension modules, which are by far the most common types.

On macOS, you'll need the command-line developer tools available through `xcode`:

Shell

```
$ xcode-select --install
```

On Windows, you'll need to install Microsoft Visual C++:

1. Open the Visual Studio downloads page in your browser.
2. Select *Tools for Visual Studio* → *Build Tools for Visual Studio* → *Download*.
3. Run the resulting `.exe` installer.
4. In the installer, select *C++ Build Tools* → *Install*.
5. Restart your machine.

On Linux, you need a compiler such as `gcc` or `g++`/`c++`.

With that squared away, you're ready to build a platform wheel for UltraJSON (`ujson`), a JSON encoder and decoder written in pure C with Python 3 bindings. Using `ujson` is a good toy example because it covers a few bases:

1. It contains an extension module, `ujson`.
2. It depends on the Python development headers to compile (`#include <Python.h>`) but is not otherwise overly complicated. `ujson` is designed to do one thing and do it well, which is to read and write JSON!

You can clone the project from GitHub, navigate into its directory, and build it:

Shell

```
$ git clone -q --branch 2.0.3 git@github.com:ultrajson/ultrajson.git
$ cd ultrajson
$ python setup.py bdist_wheel
```

You should see a whole lot of output. Here's a trimmed version on macOS, where the Clang compiler driver is used:

Text

```
clang -Wno-unused-result -Wsign-compare -Wunreachable-code -DNDEBUG -g ...
...
creating 'dist/ujson-2.0.3-cp38-cp38-macosx_10_15_x86_64.whl'
adding 'ujson.cpython-38-darwin.so'
```

The lines starting with `clang` show the actual call to the compiler complete with a trove of compilation flags. You might also see tools such as `MSVC` (Windows) or `gcc` (Linux) depending on the operating system.

If you run into a `fatal error` after executing the above code, don't worry. You can expand the box below to learn how to deal with this problem.

Fixing "fatal error: Python.h file not found" Show/Hide

If you inspect UltraJSON's `setup.py`, then you'll see that it customizes some compiler flags such as `-D_GNU_SOURCE`. The intricacies of controlling the compilation process through `setup.py` are beyond the scope of this tutorial, but you should know that it's possible to have fine-grained control over how the compiling and linking occurs.

If you look in `dist`, then you should see the created wheel:

Shell

```
$ ls dist/
ujson-2.0.3-cp38-cp38-macosx_10_15_x86_64.whl
```

Note that the name may vary based on your platform. For example, you'd see `win_amd64.whl` on 64-bit Windows.

You can peek into the wheel file and see that it contains the compiled extension:

Shell

```
$ unzip -l dist/ujson-*.whl
...
  Length      Date    Time    Name
---------  ---------- -----   ----
   105812  05-10-2020 19:47   ujson.cpython-38-darwin.so
   ...
```

This example shows an output for macOS, `ujson.cpython-38-darwin.so`, which is a shared object (`.so`) file, also called a dynamic library.

Linux: Building `manylinux` Wheels

As a package developer, you'll rarely want to build wheels for a single Linux variant. Linux wheels demand a specialized set of conventions and tools so that they can work across different Linux environments.

Unlike wheels for macOS and Windows, wheels built on one Linux variant have no guarantee of working on another Linux variant, even one with the same machine architecture. In fact, if you build a wheel on an out-of-the-box Linux container, then PyPI won't even accept that wheel if you try to upload it!

If you want your package to be available across a range of Linux clients, then you want a `manylinux` wheel. A `manylinux` wheel is a particular type of a platform wheel that is accepted by most Linux variants. It must be built in a specific environment, and it requires a tool called `auditwheel` that renames the wheel file to indicate that it's a `manylinux` wheel.

Note: Even if you're approaching this tutorial from the developer rather than the user perspective, make sure that you've read the section on the `manylinux` wheel tag before continuing with this section.

Building a `manylinux` wheel allows you to target a wider range of user platforms. PEP 513 specifies a particular (and archaic) version of CentOS with an array of Python versions available. The choice between CentOS and Ubuntu or any other distribution doesn't carry any special distinction. The point is for the build environment to consist of a stock Linux operating system with a limited set of external shared libraries that are common to different Linux variants.

Thankfully, you don't have to do this yourself. PyPA provides a set of Docker images that give you this environment with a few mouse clicks:

- **Option 1** is to run `docker` from your development machine and mount your project using a Docker volume so that it's is accessible in the container filesystem.
- **Option 2** is to use a CI/CD solution such as CircleCI, GitHub Actions, Azure DevOps, or Travis-CI, which will pull your project and run the build on an action such as a push or tag.

The Docker images are provided for the different `manylinux` flavors:

`manylinux` Tag	Architecture	Docker Image

`manylinux` Tag	Architecture	Docker Image
`manylinux1`	x86-64	quay.io/pypa/manylinux1_x86_64
`manylinux1`	i686	quay.io/pypa/manylinux1_i686
`manylinux2010`	x86-64	quay.io/pypa/manylinux2010_x86_64
`manylinux2010`	i686	quay.io/pypa/manylinux2010_i686
`manylinux2014`	x86-64	quay.io/pypa/manylinux2014_x86_64
`manylinux2014`	i686	quay.io/pypa/manylinux2014_i686
`manylinux2014`	aarch64	quay.io/pypa/manylinux2014_aarch64
`manylinux2014`	ppc64le	quay.io/pypa/manylinux2014_ppc64le
`manylinux2014`	s390x	quay.io/pypa/manylinux2014_s390x

To get started, PyPA also provides an example repository, python-manylinux-demo, which is a demo project for building `manylinux` wheels in conjunction with Travis-CI.

While it's common to build wheels as a part of a remote-hosted CI solution, you can also build `manylinux` wheels locally. To do so, you'll need Docker installed. Docker Desktop is available for macOS, Windows, and Linux.

First, clone the demo project:

Shell
```
$ git clone -q git@github.com:pypa/python-manylinux-demo.git
$ cd python-manylinux-demo
```

Next, define a few shell variables for the `manylinux1` Docker image and platform, respectively:

Shell
```
$ DOCKER_IMAGE='quay.io/pypa/manylinux1_x86_64'
$ PLAT='manylinux1_x86_64'
```

The `DOCKER_IMAGE` variable is the image maintained by PyPA for building `manylinux` wheels, hosted at Quay.io. The platform (`PLAT`) is a necessary piece of information to feed to `auditwheel`, letting it know what platform tag to apply.

Now you can pull the Docker image and run the wheel-builder script within the container:

Shell
```
$ docker pull "$DOCKER_IMAGE"
$ docker container run -t --rm \
      -e PLAT=$PLAT \
      -v "$(pwd)":/io \
      "$DOCKER_IMAGE" /io/travis/build-wheels.sh
```

This tells Docker to run the `build-wheels.sh` shell script inside the `manylinux1_x86_64` Docker container, passing `PLAT` as an environment variable available in the container. Since you used `-v` (or `--volume`) to bind-mount a volume, the wheels produced in the container will now be accessible on your host machine in the `wheelhouse` directory:

Shell

```
$ ls -1 wheelhouse
python_manylinux_demo-1.0-cp27-cp27m-manylinux1_x86_64.whl
python_manylinux_demo-1.0-cp27-cp27mu-manylinux1_x86_64.whl
python_manylinux_demo-1.0-cp35-cp35m-manylinux1_x86_64.whl
python_manylinux_demo-1.0-cp36-cp36m-manylinux1_x86_64.whl
python_manylinux_demo-1.0-cp37-cp37m-manylinux1_x86_64.whl
python_manylinux_demo-1.0-cp38-cp38-manylinux1_x86_64.whl
```

In a few short commands, you have a set of `manylinux1` wheels for CPython 2.7 through 3.8. A common practice is also to iterate over different architectures. For instance, you could repeat this process for the `quay.io/pypa/manylinux1_i686` Docker image. This would build `manylinux1` wheels targeting 32-bit (i686) architecture.

If you'd like to dive deeper into building wheels, then a good next step is to learn from the best. Start at the Python Wheels page, pick a project, navigate to its source code (on a place like GitHub, GitLab, or Bitbucket), and see for yourself how it builds wheels.

Many of the projects on the Python Wheels page are pure-Python projects and distribute universal wheels. If you're looking for more complex cases, then keep an eye out for packages that use extension modules. Here are two examples to whet your appetite:

1. **`lxml`** uses a separate build script that's invoked from within the `manylinux1` Docker container.
2. **`ultrajson`** does the same and uses GitHub Actions to call the build script.

Both of these are reputable projects that offer a great examples to learn from if you're interested in building `manylinux` wheels.

Bundling Shared Libraries

One other challenge is building wheels for packages that depend on external shared libraries. The `manylinux` images contain a prescreened set of libraries such as `libpthread.so.0` and `libc.so.6`. But what if you rely on something outside that list, such as ATLAS or GFortran?

In that case, several solutions come to the rescue:

- **`auditwheel`** will bundle external libraries into an already-built wheel.
- **`delocate`** does the same on macOS.

Conveniently, `auditwheel` is present on the `manylinux` Docker images. Using `auditwheel` and `delocate` takes just one command. Just tell them about the wheel file(s) and they'll do the rest:

Shell

```
$ auditwheel repair <path-to-wheel.whl>  # For manylinux
$ delocate-wheel <path-to-wheel.whl>  # For macOS
```

This will detect the needed external libraries through your project's `setup.py` and bundle them in to the wheel as if they were part of the project.

An example of a project that takes advantage of `auditwheel` and `delocate` is `pycld3`, which provides Python bindings for the Compact Language Detector v3 (CLD3).

The `pycld3` package depends on `libprotobuf`, which is not a commonly installed library. If you peek inside a `pycld3` macOS wheel, then you'll see that `libprotobuf.22.dylib` is included there. This is a **dynamically linked shared library** that's bundled into the wheel:

Shell

```
$ unzip -l pycld3-0.20-cp38-cp38-macosx_10_15_x86_64.whl
...
       51  04-10-2020 11:46   cld3/__init__.py
   939984  04-10-2020 07:50   cld3/_cld3.cpython-38-darwin.so
  2375836  04-10-2020 07:50   cld3/.dylibs/libprotobuf.22.dylib
---------                     -------
  3339279                     8 files
```

The wheel comes prepackaged with `libprotobuf`. A `.dylib` is similar to a Unix `.so` file or Windows `.dll` file, but I admittedly don't know the nitty-gritty of the difference beyond that.

`auditwheel` and `delocate` know to include `libprotobuf` because `setup.py` tells them to through the `libraries` argument:

Python

```python
setup(
    # ...
    libraries=["protobuf"],
    # ...
)
```

This means that `auditwheel` and `delocate` save users the trouble of installing `protobuf` as long as they're installing from a platform and Python combination that has a matching wheel.

If you're distributing a package that has external dependencies like this, then you can do your users a favor by using `auditwheel` or `delocate` to save them the extra step of installing the dependencies themselves.

Building Wheels in Continuous Integration

An alternative to building wheels on your local machine is to build them automatically within your project's CI pipeline.

There are myriad CI solutions that integrate with the major code hosting services. Among them are Appveyor, Azure DevOps, BitBucket Pipelines, Circle CI, GitLab, GitHub Actions, Jenkins, and Travis CI, to name just a few.

The purpose of this tutorial isn't to render a judgment as to which CI service is best for building wheels, and any listing of which CI services support which containers would quickly become outdated given the speed at which CI support is evolving. However, this section can help to get you started.

If you're developing a pure-Python package, the `bdist_wheel` step is a blissful one-liner: it's largely irrelevant which container OS and platform you build the wheel on. Virtually all major CI services should enable you to do this in a no-frills fashion by defining steps in a special YAML file within the project.

For example, here's the syntax you could use with GitHub Actions:

YAML

```yaml
 1 name: Python wheels
 2 on:
 3   release:
 4     types:
 5       - created
 6 jobs:
 7   wheels:
 8     runs-on: ubuntu-latest
 9     steps:
10     - uses: actions/checkout@v2
11     - name: Set up Python 3.x
12       uses: actions/setup-python@v2
13       with:
14         python-version: '3.x'
15     - name: Install dependencies
16       run: python -m pip install --upgrade setuptools wheel
17     - name: Build wheels
18       run: python setup.py bdist_wheel
19     - uses: actions/upload-artifact@v2
20       with:
21         name: dist
22         path: dist
```

In this configuration file, you build a wheel using the following steps:

1. In **line 8**, you specify that the job should run on an Ubuntu machine.
2. In **line 10**, you use the `checkout` action to set up your project repository.
3. In **line 14**, you tell the CI runner to use the latest stable version of Python 3.
4. In **line 21**, you request that the resulting wheel be available as an artifact that you can download from the UI once the job completes.

However, if you have a complex project (maybe one with C extensions or Cython code) and you're working to craft a CI/CD pipeline to automatically build wheels, then there will likely be additional steps involved. Here are a few projects through which you can learn by example:

- `yarl`
- `msgpack`
- `markupsafe`
- `cryptography`

Many projects roll their own CI configuration. However, some solutions have emerged for reducing the amount of code specified in configuration files to build wheels. You can use the cibuildwheel tool directly on your CI server to cut down on the lines of code and configuration that it takes to build multiple platform wheels. There's also multibuild, which provides a set of shell scripts for assisting with building wheels on Travis CI and AppVeyor.

Making Sure Your Wheels Spin Right

Building wheels that are structured correctly can be a delicate operation. For instance, if your Python package uses a `src` layout and you forget to specify that properly in `setup.py`, then the resulting wheel might contain a directory in the wrong place.

One check that you can use after `bdist_wheel` is the `check-wheel-contents` tool. It looks for common problems such as the package directory having an abnormal structure or the presence of duplicate files:

Shell

```shell
$ check-wheel-contents dist/*.whl
dist/ujson-2.0.3-cp38-cp38-macosx_10_15_x86_64.whl: OK
```

In this case, `check-wheel-contents` indicates that everything with the `ujson` wheel checks out. If not, `stdout` will show a summary of possible issues much like a linter such as `flake8`.

Another way to confirm that the wheel you've built has the right stuff is to use TestPyPI. First, you can upload the package there:

Shell

```
$ python -m twine upload \
      --repository-url https://test.pypi.org/legacy/ \
      dist/*
```

Then, you can download the same package for testing as if it were the real thing:

Shell

```
$ python -m pip install \
      --index-url https://test.pypi.org/simple/ \
      <pkg-name>
```

This allows you to test your wheel by uploading and then downloading your own project.

Uploading Python Wheels to PyPI

Now it's time to upload your Python package. Since a `sdist` and a wheel both get put in the `dist/` directory by default, you can upload them both using the `twine` tool, which is a utility for publishing packages to PyPI:

Shell

```
$ python -m pip install -U twine
$ python -m twine upload dist/*
```

Since both `sdist` and `bdist_wheel` output to `dist/` by default, you can safely tell `twine` to upload everything under `dist/` using a shell wildcard (`dist/*`).

Conclusion

Understanding the pivotal role that wheels play in the Python ecosystem can make your life easier as both a user and developer of Python packages. Furthermore, increasing your Python literacy when it comes to wheels will help you to better understand what's happening when you install a package and when, in increasingly rare cases, that operation goes awry.

In this tutorial, you learned:

- What wheels are and how they compare to **source distributions**
- How you can use wheels to control the **package installation** process
- What the differences are between **universal**, **pure-Python**, and **platform** wheels
- How to **create and distribute** wheels for your own Python packages

You now have a solid understanding of wheels from both a user's and a developer's perspective. You're well equipped to build you own wheels and make your project's installation process quick, convenient, and stable.

See the section below for some additional reading to dive deeper into the rapidly-expanding wheel ecosystem.

Resources

The Python Wheels page is dedicated to tracking support for wheels among the 360 most downloaded packages on PyPI. The adoption rate is pretty respectable at the time of this tutorial, at 331 out of 360, or around 91 percent.

There have been a number of Python Enhancement Proposals (PEPs) that have helped with the specification and evolution of the wheel format:

- PEP 425 - Compatibility Tags for Built Distributions

- PEP 427 - The Wheel Binary Package Format 1.0
- PEP 491 - The Wheel Binary Package Format 1.9
- PEP 513 - A Platform Tag for Portable Linux Built Distributions
- PEP 571 - The `manylinux2010` Platform Tag
- PEP 599 - The `manylinux2014` Platform Tag

Here's a shortlist of the various wheel packaging tools mentioned in this tutorial:

- `pypa/wheel`
- `pypa/auditwheel`
- `pypa/manylinux`
- `pypa/python-`manylinux-demo
- `jwodder/check-wheel-contents`
- `matthew-brett/delocate`
- `matthew-brett/multibuild`
- `joerick/cibuildwheel`

The Python documentation has several articles covering wheels and source distributions:

- Generating Distribution Archives
- Creating a Source Distribution

Finally, here are a few more useful links from PyPA:

- Packaging your Project
- An Overview of Packaging for Python