Quickstart

Installation

You can install the latest version of `setuptools` using pip:

```
pip install --upgrade setuptools
```

Most of the times, however, you don't have to...

Instead, when creating new Python packages, it is recommended to use a command line tool called build. This tool will automatically download `setuptools` and any other build-time dependencies that your project might have. You just need to specify them in a `pyproject.toml` file at the root of your package, as indicated in the following section.

You can also install build using pip:

```
pip install --upgrade build
```

This will allow you to run the command: `python -m build`.

> **Important**
>
> Please note that some operating systems might be equipped with the `python3` and `pip3` commands instead of `python` and `pip` (but they should be equivalent). If you don't have `pip` or `pip3` available in your system, please check out pip installation docs.

Every python package must provide a `pyproject.toml` and specify the backend (build system) it wants to use. The distribution can then be generated with whatever tool that provides a `build sdist`-like functionality.

Basic Use

When creating a Python package, you must provide a `pyproject.toml` file containing a `build-system` section similar to the example below:

```toml
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

This section declares what are your build system dependencies, and which library will be used to actually do the packaging.

> **Note**
>
> Historically this documentation has unnecessarily listed `wheel` in the `requires` list, and many projects still do that. This is not recommended. The backend automatically adds `wheel` dependency when it is required, and listing it explicitly causes it to be unnecessarily required for source distribution builds. You should only include `wheel` in `requires` if you need to explicitly access it during build time (e.g. if your project needs a `setup.py` script that imports `wheel`).

In addition to specifying a build system, you also will need to add some package information such as metadata, contents, dependencies, etc. This can be done in the same `pyproject.toml` file, or in a separated one: `setup.cfg` or `setup.py` [1].

The following example demonstrates a minimum configuration (which assumes the project depends on requests and importlib-metadata to be able to run):

pyproject.toml **setup.cfg** **setup.py** [1]

```python
from setuptools import setup

setup(
    name='mypackage',
    version='0.0.1',
    install_requires=[
        'requests',
        'importlib-metadata; python_version == "3.8"',
    ],
)
```

See Keywords for more information.

Finally, you will need to organize your Python code to make it ready for distributing into something that looks like the following (optional files marked with `#`):

```
mypackage
├── pyproject.toml  # and/or setup.cfg/setup.py (depending on the configuration method)
|   # README.rst or README.md (a nice description of your package)
|   # LICENCE (properly chosen license information, e.g. MIT, BSD-3, GPL-3, MPL-2, etc...)
└── mypackage
    ├── __init__.py
    └── ... (other Python files)
```

With build installed in your system, you can then run:

```
python -m build
```

You now have your distribution ready (e.g. a `tar.gz` file and a `.whl` file in the `dist` directory), which you can upload to PyPI!

Of course, before you release your project to PyPI, you'll want to add a bit more information to help people find or learn about your project. And maybe your project will have grown by then to include a few dependencies, and perhaps some data files and scripts. In the next few sections, we will walk through the additional but essential information you need to specify to properly package your project.

> **Info: Using `setup.py`**
>
> Setuptools offers first class support for `setup.py` files as a configuration mechanism.
>
> It is important to remember, however, that running this file as a script (e.g. `python setup.py sdist`) is strongly **discouraged**, and that the majority of the command line interfaces are (or will be) **deprecated** (e.g. `python setup.py install`, `python setup.py bdist_wininst`, ...).
>
> We also recommend users to expose as much as possible configuration in a more *declarative* way via the pyproject.toml or setup.cfg, and keep the `setup.py` minimal with only the dynamic parts (or even omit it completely if applicable).
>
> See Why you shouldn't invoke setup.py directly for more background.

Overview

Package discovery

For projects that follow a simple directory structure, `setuptools` should be able to automatically detect all packages and namespaces. However, complex projects might include additional folders and supporting files that not necessarily should be distributed (or that can confuse `setuptools` auto discovery algorithm).

Therefore, `setuptools` provides a convenient way to customize which packages should be distributed and in which directory they should be found, as shown in the example below:

pyproject.toml **setup.cfg** **setup.py** [1]

```python
from setuptools import setup, find_packages  # or find_namespace_packages

setup(
    # ...
    packages=find_packages(
        # All keyword arguments below are optional:
        where='src',  # '.' by default
        include=['mypackage*'],  # ['*'] by default
        exclude=['mypackage.tests'],  # empty by default
    ),
    # ...
)
```

When you pass the above information, alongside other necessary information, `setuptools` walks through the directory specified in `where` (defaults to `.`) and filters the packages it can find

following the `include` patterns (defaults to `*`), then it removes those that match the `exclude` patterns (defaults to empty) and returns a list of Python packages.

For more details and advanced use, go to Package Discovery and Namespace Packages.

> **Tip**
>
> Starting with version 61.0.0, setuptools' automatic discovery capabilities have been improved to detect popular project layouts (such as the flat-layout and src-layout) without requiring any special configuration. Check out our reference docs for more information.

Entry points and automatic script creation

Setuptools supports automatic creation of scripts upon installation, that run code within your package if you specify them as entry points. An example of how this feature can be used in `pip`: it allows you to run commands like `pip install` instead of having to type `python -m pip install`.

The following configuration examples show how to accomplish this:

pyproject.toml | setup.cfg | setup.py [1]

```python
setup(
    # ...
    entry_points={
        'console_scripts': [
            'cli-name = mypkg.mymodule:some_func',
        ]
    }
)
```

When this project is installed, a `cli-name` executable will be created. `cli-name` will invoke the function `some_func` in the `mypkg/mymodule.py` file when called by the user. Note that you can also use the `entry-points` mechanism to advertise components between installed packages and implement plugin systems. For detailed usage, go to Entry Points.

Dependency management

Packages built with `setuptools` can specify dependencies to be automatically installed when the package itself is installed. The example below shows how to configure this kind of dependencies:

pyproject.toml | setup.cfg | setup.py [1]

```python
setup(
    # ...
    install_requires=["docutils", "requests <= 0.4"],
    # ...
)
```

Each dependency is represented by a string that can optionally contain version requirements (e.g. one of the operators <, >, <=, >=, == or !=, followed by a version identifier), and/or conditional

environment markers, e.g. `sys_platform == "win32"` (see Version specifiers for more information).

When your project is installed, all of the dependencies not already installed will be located (via PyPI), downloaded, built (if necessary), and installed. This, of course, is a simplified scenario. You can also specify groups of extra dependencies that are not strictly required by your package to work, but that will provide additional functionalities. For more advanced use, see Dependencies Management in Setuptools.

Including Data Files

Setuptools offers three ways to specify data files to be included in your packages. For the simplest use, you can simply use the `include_package_data` keyword:

pyproject.toml | setup.cfg | setup.py [1]

```python
setup(
    # ...
    include_package_data=True,
    # ...
)
```

This tells setuptools to install any data files it finds in your packages. The data files must be specified via the MANIFEST.in file or automatically added by a Revision Control System plugin. For more details, see Data Files Support.

Development mode

`setuptools` allows you to install a package without copying any files to your interpreter directory (e.g. the `site-packages` directory). This allows you to modify your source code and have the changes take effect without you having to rebuild and reinstall. Here's how to do it:

```
pip install --editable .
```

See Development Mode (a.k.a. "Editable Installs") for more information.

> **Tip**
>
> Prior to pip v21.1, a `setup.py` script was required to be compatible with development mode. With late versions of pip, projects without `setup.py` may be installed in this mode.
>
> If you have a version of `pip` older than v21.1 or is using a different packaging-related tool that does not support **PEP 660**, you might need to keep a `setup.py` file in file in your repository if you want to use editable installs.
>
> A simple script will suffice, for example:
>
> ```python
> from setuptools import setup
>
> setup()
> ```
>
> You can still keep all the configuration in pyproject.toml and/or setup.cfg

Uploading your package to PyPI

After generating the distribution files, the next step would be to upload your distribution so others can use it. This functionality is provided by twine and is documented in the Python packaging tutorial.

Transitioning from `setup.py` to `setup.cfg`

To avoid executing arbitrary scripts and boilerplate code, we are transitioning into a full-fledged `setup.cfg` to declare your package information instead of running `setup()`. This inevitably brings challenges due to a different syntax. Here we provide a quick guide to understanding how `setup.cfg` is parsed by `setuptools` to ease the pain of transition.

Resources on Python packaging

Packaging in Python can be hard and is constantly evolving. Python Packaging User Guide has tutorials and up-to-date references that can help you when it is time to distribute your work.

Notes

[1] (1,2,3,4,5,6) New projects are advised to avoid `setup.py` configurations (beyond the minimal stub) when custom scripting during the build is not necessary. Examples are kept in this document to help people interested in maintaining or contributing to existing packages that use `setup.py`. Note that you can still keep most of configuration declarative in setup.cfg or pyproject.toml and use `setup.py` only for the parts not supported in those files (e.g. C extensions). See note.