editables

Release 0.5

Paul Moore

Jan 26, 2024

CONTENTS

CHAPTER

ONE

BASIC WORKFLOW

The `editables` project is designed to support *build backends*, allowing them to declare what they wish to expose as "editable", and returning a list of support files that need to be included in the wheel generated by the `build_editable` backend hook. Note that the `editables` library does not build wheel files directly - it returns the content that needs to be added to the wheel, but it is the build backend's responsibility to actually create the wheel from that data.

1.1 Create a project

The first step is for the backend to create an "editable project". The project name must follow the normal rules for Python project names from PEP 426.

```
project = EditableProject("myproject")
```

1.2 Specify what to expose

Once the project has been created, the backend can specify which files should be exposed when the editable install is done. There are two mechanisms currently implemented for this.

1.2.1 Adding a directory to `sys.path`

To add a particular directory (typically the project's "src" directory) to `sys.path` at runtime, simply call the `add_to_path` method

```
project.add_to_path("src")
```

This will simply write the given directory into the `.pth` file added to the wheel. See the "Implementation Details" section for more information. Note that this method requires no runtime support.

1.2.2 Adding a directory as package content

To expose a directory as a package on `sys.path`, call the `add_to_subpackage` method, giving the package name to use, and the path to the directory containing the contents of that package.

For example, if the directory `src` contains a package `my_pkg`, which you want to expose to the target interpreter as `some.package.my_pkg`, run the following:

```
project.add_to_subpackage("some.package", "src")
```

Note that everything in the source directory will be available under the given package name, and the source directory should *not* contain an `__init__.py` file (if it does, that file will simply be ignored).

Also, the target (`some.package` here) must *not* be an existing package that is already part of the editable wheel. This is because its `__init__.py` file will be overwritten by the one created by this method.

CHAPTER

TWO

MAPPING INDIVIDUAL FILES/PACKAGES

To expose a single `.py` file as a module, call the `map` method, giving the name by which the module can be imported, and the path to the implementation `.py` file. It *is* possible to give the module a name that is not the same as the implementation filename, although this is expected to be extremely uncommon.

```
project.map("module", "src/module.py")
```

To expose a directory with an `__init__.py` file as a package, the `map` method is used in precisely the same way, but with the directory name:

```
project.map("mypackage", "src/mypackage")
```

The directory *must* be a Python package - i.e., it must contain an `__init__.py` file, and the target package name must be a top-level name, not a dotted name.

Using the `map` method does require a runtime support module.

2.1 Build the wheel

2.1.1 Files to add

Once all of the content to expose is specified, the backend can start building the wheel. To determine what files to write to the wheel, the `files` method should be used. This returns a sequence of pairs, each of which specifies a filename, and the content to write to that file. Both the name and the content are strings, and so should be encoded appropriately (i.e., in UTF-8) when writing to the wheel.

```
for name, content in my_project.files():
    wheel.add_file(name, content)
```

Note that the files to be added must be included unchanged - it is *not* supported for the caller to modify the returned content. Also, it is the caller's responsibility to ensure that none of the generated files clash with files that the caller is adding to the wheel as part of its own processes.

2.1.2 Runtime dependencies

If the `map` method is used, the resulting wheel will require that the runtime support module is installed. To ensure that is the case, dependency metadata must be added to the wheel. The `dependencies` method provides the required metadata.

```
for dep in my_project.dependencies():
    wheel.metadata.dependencies.add(dep)
```

Note that if the backend only uses the `add_to_path` method, no runtime support is needed, so the `dependencies` method will return an empty list. For safety, and to protect against future changes, it should still be called, though.

CHAPTER

THREE

IMPLEMENTATION DETAILS

The key feature of a project that is installed in "editable mode" is that the code for the project remains in the project's working directory, and what gets installed into the user's Python installation is simply a "pointer" to that code. The implication of this is that the user can continue to edit the project source, and expect to see the changes reflected immediately in the Python interpreter, without needing to reinstall.

The exact details of how such a "pointer" works, and indeed precisely how much of the project is exposed to Python, are generally considered to be implementation details, and users should not concern themselves too much with how things work "under the hood". However, there are practical implications which users of this library (typically build backend developers) should be aware of.

The basic import machinery in Python works by scanning a list of directories recorded in `sys.path` and looking for Python modules and packages in these directories. (There's a *lot* more complexity behind the scenes, and interested readers are directed to the Python documentation for more details). The initial value of `sys.path` is set by the interpreter, but there are various ways of influencing this.

As part of startup, Python checks various "site directories" on `sys.path` for files called `*.pth`. In their simplest form, `.pth` files contain a list of directory names, which are *added* to `sys.path`. In addition, for more advanced cases, `.pth` files can also run executable code (typically, to set up import hooks to further configure the import machinery).

3.1 Editables using `.pth` entries

The simplest way of setting up an editable project is to install a `.pth` file containing a single line specifying the project directory. This will cause the project directory to be added to `sys.path` at interpreter startup, making it available to Python in "editable" form.

This is the approach which has been used by setuptools for many years, as part of the `setup.py develop` command, and subsequently exposed by pip under the name "editable installs", via the command `pip install --editable <project_dir>`.

In general, this is an extremely effective and low-cost approach to implementing editable installs. It does, however, have one major disadvantage, in that it does *not* necessarily expose the same packages as a normal install would do. If the project is not laid out with this in mind, an editable install may expose importable files that were not intended. For example, if the project root directory is added directly to the `.pth` file, `import setup` could end up running the project's `setup.py`! However, the recommended project layout, putting the Python source in a `src` subdirectory (with the `src` directory then being what gets added to `sys.path`) reduces the risk of such issues significantly.

The `editables` project implements this approach using the `add_to_path` method.

3.2 Package-specific paths

If a package sets the `__path__` variable to a list of those directories, the import system will search those directories when looking for subpackages or submodules. This allows the user to "graft" a directory into an existing package, simply by setting an appropriate `__path__` value.

The `editables` project implements this approach using the `add_to_subpackage` method.

3.3 Import hooks

Python's import machinery includes an "import hook" mechanism which in theory allows almost any means of exposing a package to Python. Import hooks have been used to implement importing from zip files, for example. It is possible, therefore, to write an import hook that exposes a project in editable form.

The `editables` project implements an import hook that redirects the import of a package to a filesystem location specifically designated as where that package's code is located. By using this import hook, it is possible to exercise precise control over what is exposed to Python. For details of how the hook works, readers should investigate the source of the `editables.redirector` module, part of the `editables` package.

The `editables` project implements this approach for the `map` method. The `.pth` file that gets written loads the redirector and calls a method on it to add the requested mappings to it.

There are two downsides to this approach, as compared to the simple `.pth` file mechanism - lack of support for implicit namespace packages, and the need for runtime support code.

The first issue (lack of support for implicit namespace packages) is unfortunate, but inherent in how Python (currently) implements the feature. Implicit namespace package support is handled as part of how the core import machinery does directory scans, and does not interact properly with the import hook mechanisms. As a result, the `editables` import hook does not support implicit namespace packages, and will probably never be able to do so without help from the core Python implementation[1].

The second issue (the need for runtime support) is more of an inconvenience than a major problem. Because the implementation of the import hook is non-trivial, it should be shared between all editable installs, to avoid conflicts between import hooks, and performance issues from having unnecessary numbers of identical hooks running. As a consequence, projects installed in this manner will have a runtime dependency on the hook implementation (currently distributed as part of `editables`, although it could be split out into an independent project).

3.4 Reserved Names

The `editables` project uses the following file names when building an editable wheel. These should be considered reserved. While backends would not normally add extra files to wheels generated using this library, they are allowed to do so, as long as those files don't use any of the reserved names.

1. `<project_name>.pth`
2. `_editable_impl_<project_name>*.py`

Here, `<project_name>` is the name supplied to the `EditableProject` constructor, normalised as described in PEP 503, with dashes replaced by underscores.

---

[1] The issue is related to how the same namespace can be present in multiple `sys.path` entries, and must be dynamically recomputed if the filesystem changes while the interpreter is running.

CHAPTER

FOUR

USE CASES

We will cover here the main supported use cases for editable installs, including the recommended approaches for exposing the files to the import system.

4.1 Project directory installed "as is"

A key example of this is the recommended "`src` layout" for a project, where a single directory (typically named `src`) is copied unchanged into the target site-packages.

For this use case, the `project.add_to_path` method is ideal, making the project directory available to the import system directly.

There are almost no downsides to this approach, as it is using core import system mechanisms to manage `sys.path`. Furthermore, the method is implemented using `.pth` files, which are recognised by static analysis tools such as type checkers, and so editable installs created using this method will be visible in such tools.

4.2 Project directory installed under an explicit package name

This is essentially the same as the previous use case, but rather than installing the project directory directly into site-packages, it is installed under a partocular package name. So, for example, if the project has a `src` directory containing a package `foo` and a module `bar.py`, the requirement is to install the contents of `src` as `my.namespace.foo` and `my.namespace.bar`.

For this use case, the `project.add_to_subpackage` method is available. This method creates the `my.namespace` package (by installing an `__init__.py` file for it into site-packages) and gives that package a `__path__` attribute pointing to the source directory to be installed under that package name.

Again, this approach uses core import system mechanisms, and so will have few or no downsides at runtime. However, because this approach relies on *runtime* manipulation of `sys.path`, it will not be recognised by static analysis tools.

4.3 Installing part of a source directory

The most common case for this is a "flat" project layout, where the package and module files to be installed are stored alongside project files such as `pyproject.toml`. This layout is typically *not* recommended, particularly for new projects, although older projects may be using this type of layout for historical reasons.

The core import machinery does not provide a "native" approach supporting excluding part of a directory like this, so custom import hooks are needed to implement it. At the time of writing, all such custom hook implementations have limitations, and should be considered experimental. As a result, build backends should *always* prefer one of the other implementation methods when available.

The `project.map` method allows mapping of either a single Python file, or a Python package directory, to an explicit top-level name in the import system. It does this by installing a `.pth` file and a Python module. The `.pth` file simply runs the Python module, and the module installs the requested set of mappings using an import hook exported by the `editables` module.

Downsides of this approach are:

1. The approach depends on the ability to run executable code from a `.pth` file. While this is a supported capability of `.pth` files, it is considered a risk, and there have been proposals to remove it. If that were to happen, this mechanism would no longer work.
2. It adds a *runtime* dependency on the `editables` module, rather than just a build-time dependency.
3. The import hook has known limitations when used with implicit namespace packages - there is a CPython issue discussing some of the problems.

4.4 Unsupported use cases

In addition to the above there are a number of use cases which are explicitly **not** supported by this library. That is not to say that editable installs cannot do these things, simply that the build backend will need to provide its own support.

4.4.1 Metadata changes

This library does not support dynamically changing installed project metadata when the project source changes. Typically, a reinstall is needed in those cases. A significant example of a metadata change is a change to the script entry points, which affects what command-line executables are installed.

4.4.2 Binary extensions

Binary extensions require a build step when the source code is changed. This library does not support any sort of automatic rebuilding, nor does it support automatic reinstallation of binaries.

The build backend may choose to expose the "working" version of the built binary, for example by placing a symbolic link to the binary in a directory that is visible to the import system as a result of `project.add_to_path`, but that would need to be implemented by the backend.

4.4.3 Mapping non-Python directories or files

The methods of an editable project are all intended explicitly for exposing *Python code* to the import system. Other types of resource, such as data files, are *not* supported, except in the form of package data physically located in a Python package directory in the source.

4.4.4 Combining arbitrary code into a package

The library assumes that a typical project layout, at least roughly, matches the installed layout - and in particular that Python package directories are "intact" in the source. Build backends can support more complex structures, but in order to expose them as editable installs, they need to create some form of "live" reflection of the final layout in a local directory (for example by using symbolic links) and create the editable install using that shadow copy of the source.

It is possible that a future version of this library may add support for more complex mappings of this form, but that would likely require a significant enhancement to the import hook mechanism being used, and would be a major, backward incompatible, change. There are currently no plans for such a feature, though.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search