From Script to Shared Resource

A Modern Guide to Packaging in R and Python for Researchers

2026-05-06

Part 1

Foundations of Packaging

Language-agnostic principles for scientific software distribution.

Why Package Scientific Software?

Ease of Installation

Allow users to install your tool with a single command rather than manual configuration. Significantly lowers the barrier for others to adopt your research methods.

Dependency Management

Automate the process of resolving and installing the libraries your code relies on. This prevents version conflicts and ensures the software environment is consistently correct.

Accessibility

Published packages are findable via central repositories, making your work part of the global research infrastructure. It moves code from a private folder to a public, citeable asset.

Standardization

Following community standards makes your project instantly familiar to other researchers. It allows automated tools to build, test, and analyze your code without custom tweaks.

Reproducibility

Packaging captures the specific state of code and dependencies at a point in time. This is critical for scientific integrity, ensuring colleagues can replicate your exact results.

Pre-packaging checklist

- ✅ **Licence**: Chosen an appropriate licence for your intended use case (Software lifecycle course materials)
- ✅ **Generalization**: Abstract your logic away from your specific machine's file paths or local input data. (Software Design Course coming soon!)
- ✅ **Documentation**: Ensure the code has a clear guide for installation and usage (README) and is documented internally (docstrings). (Documentation course materials)
- ✅ **Testing**: Implement automated tests to verify that your code still works correctly with different inputs and across a variety of environments. (Testing and CI course materials)

The Anatomy of a Software Package (R & Python)

Mandatory for CRAN

Mandatory for PyPi

Source code

Core functionality

Metadata

Essential package details: name, version, authors, dependencies

README

Overview, installation steps, and usage instructions.

Licence

Defines how others can reuse the software

Tests

Validation suite to ensure ongoing reliability

Code of Conduct

Outlines community guidelines for behaviour and standards for contributors

Additional Docs

Long-form documentation often in the form of a separate website

Resources

Example data or any other resources

Part 2

Creating and Publishing a Python Package

Follow-along live Demo

Standard Project Structure

```
mypackage
├── LICENSE.md
├── pyproject.toml
├── README.md
└── src
    └── mypackage
        ├── __init__.py
        └── mymodule.py
```

Structure: Python code lives in `src` under a nested folder with the package name. Metadata lives at the top level.

Modules: .py files that are designed to be imported; in contrast to scripts (.py files intended to be run directly)

`__init__.py`: A (not necessarily) empty file that tells Python this folder is a package. Without it, you cannot import your code. Also useful for customizing imports

Step 1: Creating the package skeleton

- uv has a helper command to setup a skeleton package structure + barebones pyproject.toml + git repository + file tracking Python version
- Suffixing with username to create unique package names for uploading to TestPyPi
- The top level folder name is the PyPi name (can't contain underscores) but the `src/mypackage` name is the import name (can't contain hyphens)

  E.g.: pip install scikit-learn vs import sklearn

```
$ uv init --package mypackage-abc123
$ tree -a mypackage-abc123
mypackage-abc123
├── .git
│   └── …
├── .gitignore
├── pyproject.toml
├── .python-version
├── README.md
└── src
    └── mypackage_abc123
        └── __init__.py
```

Step 1: Creating the package skeleton

- uv has a helper command to setup a skeleton + git repository + file to track Python version
- Suffixing with username to create unique package names for uploading to TestPyPi
- The top level folder name is the PyPi name (can't contain underscores) but the `src/mypackage` name is the import name (can't contain hyphens)
  E.g.: pip install scikit-learn vs import sklearn

```
$ uv init --package mypackage-abc123
$ tree -a mypackage-abc123
mypackage-abc123
├── .git
│   └── …
├── .gitignore
├── pyproject.toml
├── .python-version
├── README.md
└── src
    └── mypackage_abc123
        └── __init__.py
```

Task: create a package skeleton
NB: if using `pip` will need to create structure manually
NB: replace `abc123` with your username or a random string

Step 2: Populating pyproject.toml

```toml
[project]
name = "mypackage-abc123"
version = "0.1.0"
description = "Add your description
here"
readme = "README.md"
authors = [
    { name = "YOUR NAME", email =
"YOUR_EMAIL@DOMAIN" }
]
requires-python = ">=3.13"
dependencies = []

[project.scripts]
mypackage-abc123 =
"mypackage_abc123:main"

[build-system]
requires = ["uv_build>=0.8.4,<0.9.0"]
build-backend = "uv_build"
```

uv provides a skeleton configuration

authors will be populated from git if setup

dependencies will contain a list of required packages (to be updated manually or via uv add)

project.scripts specifies CLI entry-points

```
$ mypackage
```

vs

```
$ python -c "import mypackage; mypackage.main()"
```

uv provides its own build system (as of July 2025)
hatchling is another popular choice

Step 2: Populating pyproject.toml

```toml
[project]
name = "mypackage-abc123"
version = "0.1.0"
description = "Add your description
here"
readme = "README.md"
authors = [
    { name = "YOUR NAME", email =
"YOUR_EMAIL@DOMAIN }
]
requires-python = ">=3.13"
dependencies = []

[project.scripts]
mypackage-abc123 =
"mypackage_abc123:main"

[build-system]
requires = ["uv_build>=0.8.4,<0.9.0"]
build-backend = "uv_build"
```

uv provides a skeleton configuration set to version 0.1.0

authors will be populated from git if configured

dependencies will contain a list of required packages (to be updated manually or via `uv add`)

project.scripts specifies CLI entry-points

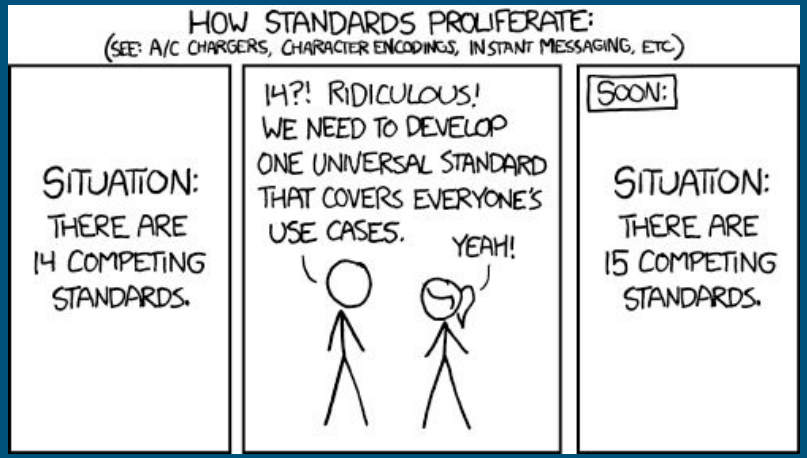
`$ mypackage` vs `$ python -c "import mypackage; mypackage.main()"`

uv provides its own build system (as of July 2025)
hatchling is another popular choice

Task: populate pyproject.toml (use anonymous name & email if don't want to be publicly visible on TestPyPi)

Digression: the evolution of Python packaging

- distutils & setup.py
- setuptools (extended distutils) & eggs
- pip
- Virtual environments
- Wheel format (replaced eggs)
- Pyproject.toml (replaced setup.py)
- 3rd party tools:
  - Poetry
  - Pyvenv
  - PDM
  - And many others!



https://xkcd.com/927

Step 3: Developing

Modules must be placed under `src/mypackage_abc123`

Test functionality by importing in a script external to the package (`uv run python demo.py`), or by using unit tests (Test Driven Development, not shown)

No need to tell `uv` when we make changes - `uv run` auto-syncs the venv. `pip` users will need to run `pip install -e .` once first to create an 'editable install' (ideally from within a virtual environment)

```
$ tree mypackage-abc123
mypackage-abc123
├── …
├── demo.py
└── src
    └── mypackage_abc123
        └── __init__.py
        └── mymodule.py
```

```python
#mymodule.py

def test(a, b):
    print(f"a * b = {a*b}")
```

```python
#demo.py

from mypackage_abc123.mymodule import test

test(5, 3)
```

Step 3: Developing

Modules must be placed under `src/mypackage_abc123`

Test functionality by importing in a script external to the package (`uv run python demo.py`), or by using unit tests (Test Driven Development, not shown)

No need to tell `uv` when we make changes - `uv run` auto-syncs the venv. `pip` users will need to run `pip install -e .` once first to create an 'editable install' (ideally from within a virtual environment)

Task: add a module to perform some basic logic, run it, make changes and run again

```
$ tree mypackage-abc123
mypackage-abc123
├── …
├── demo.py
└── src
    └── mypackage_abc123
        └── __init__.py
        └── mymodule.py
```

```python
#mymodule.py

def test(a, b):
    print(f"a * b = {a*b}")
```

```python
#demo.py

from mypackage_abc123.mymodule import test

test(5, 3)
```

Step 3: Developing

Modules must be placed under `src/mypackage_abc123`

Test functionality by importing in a script external to the package (`uv run python demo.py`), or by using unit tests (Test Driven Development, not shown)

No need to tell `uv` when we make changes - `uv run` auto-syncs the venv. `pip` users will need to run `pip install -e .` once first to create an 'editable install' (ideally from within a virtual environment)

```
$ tree mypackage-abc123
mypackage-abc123
├── …
├── demo.py
└── src
    └── mypackage_abc123
        └── __init__.py
        └── mymodule.py
```

```python
#mymodule.py

def test(a, b):
    print(f"a * b = {a*b}")
```

```python
#demo.py

from mypackage_abc123.mymodule import test

test(5, 3)
```

Task: add a module to perform some basic logic, run it, make changes and run again

Task 2: Add a dependency (i.e. numpy) to your module and get it working (hint: you'll need to use `uv add`). Look how pyproject.toml and uv.lock change

Step 4: Publishing to (Test) PyPi

TestPyPi is a version of PyPi that is solely for testing purposes
NB: Must setup 2FA + Create & save API token via Account Settings

1: Build package into wheel (`.whl`) + copy of source (`.tar.gz`)

2: Publish to Test PyPi (defaults to regular PyPi if URL isn't provided)

uv

```
$ uv build
Building source distribution (uv build backend)...
Building wheel from source distribution (uv build backend)...
Successfully built dist/mypackage_sl561-0.1.0.tar.gz
Successfully built dist/mypackage_sl561-0.1.0-py3-none-any.whl
$ uv publish --publish-url https://test.pypi.org/legacy/
```

vanilla Python

```
# Install 'build' and 'twine' packages first
$ python -m build
…
$ python -m twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Step 4: Publishing to (Test) PyPi

TestPyPi is a version of PyPi that is solely for testing purposes
NB: Must setup 2FA + Create API token via Account Settings

1: Build package into wheel (`.whl`) + copy of source (`.tar.gz`)

2: Publish to Test PyPi (defaults to regular PyPi if URL isn't provided)

uv

```
$ uv build
Building source distribution (uv build backend)...
Building wheel from source distribution (uv build backend)...
Successfully built dist/mypackage_sl561-0.1.0.tar.gz
Successfully built dist/mypackage_sl561-0.1.0-py3-none-any.whl
$ uv publish --publish-url https://test.pypi.org/legacy/
```

vanilla Python

```
# Install 'build' and 'twine' packages first
$ python -m build
…
$ python -m twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Task: upload your package to TestPyPi

Additional Considerations

- **Linting & formatting**: Use tools like `ruff` or `black` to maintain a consistent coding style
- **Docs site**: PyPI does not host documentation pages like CRAN. You can generate a docs site using `sphinx` and host it via **GitHub Pages** or **ReadTheDocs**
- **Citing your software**: Add a `CITATION.cff` file so other researchers can easily cite your code in their publications
- **Continuous Integration (CI)**: Use GitHub Actions to automatically run your tests every time you push code, apply linting & formatting, push releases to PyPI, etc…
- **Refactoring imports**: import functions into top level package namespace in __init__.py

Part 3

Writing and Publishing an R Package

Follow-along live demo