Python Packaging

A Recap of the PYOPP Workshop

Anno Knierim

aknierim

July 25, 2025

Why Even Bother With Packaging?

Packages allow you to share your code, so other people can use it.

But also...

- Helps you keeping your code from breaking
- Benefits other people that may have faced a similar problem
- Saves time because code can be reused easily

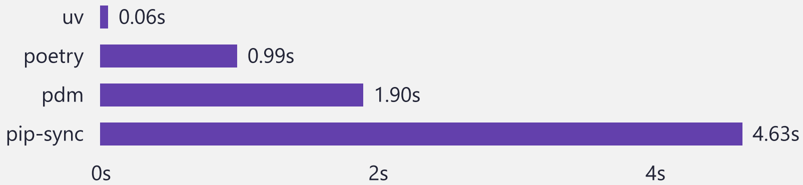
Before We Start: Package and Environment Managers

pip The standard package installer for Python. `pip` is able to install directly from PyPI and other indexes.

mamba Fast and robust, with cross-platform support. Written in C++ . Allows you to manage multiple, isolated environments. `mamba` installs from local or remote package repositories, e. g., channels.

poetry Package installer that is also able to create its own virtual environments. Handles dependency resolving better than pip. Works nicely with `pyproject.toml` files. Allows the use of lock files.

uv A new and fast package manager written in `Rust`. Can create virtual environments, and solves dependencies better and faster than pip. Allows the use of lock files.



[docs.astral.sh/uv]

Packaging: The Basics

What Even is a Package?

Import Package Any Python module that you can ***import*** using the `import` statement.

Namespace Package Packages that allow you to ***unify*** two packages with the ***same*** name.

Distribution Package An archive containing a ***collection*** of import packages combined with ***metadata*** such as dependencies.

What Even is a Package?

Import Package Any Python module that you can ***import*** using the `import` statement.

Namespace Package Packages that allow you to ***unify*** two packages with the ***same*** name.

Distribution Package An archive containing a ***collection*** of import packages combined with ***metadata*** such as dependencies.

When people talk about packages, they usually mean **distribution packages.**

How Does Python Find Installed Packages?

Example: NumPy

```
$ python -c "import numpy; print(numpy.__path__[0])"
/home/anno/.local/conda/envs/pyopp_recap/lib/python3.12/site-packages/numpy

$ ls -C $(python -c "import numpy; print(numpy.__path__[0])") | sort
_array_api_info.py     doc                     __init__.py        py.typed
_array_api_info.pyi    dtypes.py               __init__.pyi       random
char                   dtypes.pyi              lib                rec
__config__.py          exceptions.py           linalg             strings
__config__.pyi         exceptions.pyi          ma                 testing
_configtool.py         _expired_attrs_2_0.py   matlib.py          tests
_configtool.pyi        _expired_attrs_2_0.pyi  matlib.pyi         _typing
conftest.py            f2py                    matrixlib          typing
_core                  fft                     polynomial         _utils
core                   _globals.py             __pycache__        version.py
ctypeslib              _globals.pyi            _pyinstaller       version.pyi
_distributor_init.pyi  __init__.pxd            _pytesttester.pyi
_distributor_init.py   __init__.cython-30.pxd  _pytesttester.py
```

How Do I Create a Package?

There is not “just one way” to create packages, but...

- Modern packaging uses a scaffolding called `pyproject.toml` with three important sections:

  `[build-system]` Allows you to describe what build backend to use.

  `[project]` Sets up metadata for the package, such as the name or version.

  `[tool]` A section for tool configuration.

- An easy way to set up that scaffolding: `hatch`

```
$ uv pip install hatch
$ mamba install hatch
```

How Do I Create a Package?

- Use `hatch`'s CLI tool to quickstart creating a package:

```
$ hatch new my_package
```

- Let's see what this created:

```
$ head my_package/pyproject.toml
```

- You can also upgrade an existing project to use `hatch`:

```
$ hatch new --init
```

- Have a look at the Writing your `pyproject.toml` guide to learn how to customise the `pyproject.toml` file

Output

```
my_package
├── src
│   └── my_package
│       ├── __about__.py
│       └── __init__.py
├── tests
│   └── __init__.py
├── LICENSE.txt
├── README.md
└── pyproject.toml
```

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "my_package"
dynamic = ["version"]
description = ''
readme = "README.md"
requires-python = ">=3.8"
```

Dependencies

- Dependencies for your project are defined with the `dependencies` key inside the `[project]` section
- You can set dependency specifiers (aka constraints) such as versions

- Define your optional dependencies in the `[project.optional-dependencies]` section and group them
- Install optional dependencies using

```
$ uv pip install my_package[plot]
$ uv pip install "my_package[plot]"
```

Example

```
[project]
dependencies = [
  "numpy",
  "astropy≤6.1.0",
  "tomli;python_version<'3.11'",
]

[project.optional-dependencies]
plot = ["matplotlib"]
```

Dependency Groups

- Fairly new (accepted `2024-10-10`): PEP 735 Dependency Groups
- Optional dependencies that are ***not*** installed when a ***user*** installs the package, e. g., via PyPI
  - → Prevent users from installing dev tools
- Install the groups from within your source repo:

```
$ uv pip install --group dev
```

Example

```
[dependency-groups]
tests = ["pytest", "pytest-cov"]
docs = ["sphinx"]
dev = [
  "jupyter",
  "pre-commit",
  {include-group = "tests"},
  {include-group = "docs"},
]
```

Packaging: The Fun Stuff

CLI Scripts

- We can expose scripts in our package using the `pyproject.toml` `[project.scripts]` section
- Similarly: Entry points, that allow the creation of plugins, and cross-platform compatibility
  - → See ↗ Entry Points

Example

src/my_package/cli.py:

```python
def print_message():
    print("Hello World!")
    raise SystemExit(1)
```

pyproject.toml:

```toml
[project.scripts]
hello-world =
    "my_package.cli:print_message"
```

Result

```
$ hello-world
Hello World!
```

Versioning

Remember:

- `pyproject.toml` has required fields:

  ```toml
  [project]
  name = "my_package"
  version = "0.1.0"
  ```

- One way to get this version is with `hatch`

  ```
  $ hatch version
  0.1.0
  ```

- We can also set a new version using `hatch`:

  ```
  $ hatch version 0.2.0
  Old: 0.1.0
  New: 0.2.0
  ```

Static Versions *kinda* Suck

I guess...

So Let's Do Something About It

Approach #1:

- We can set the `version` field to dynamic…
- …and set the version as `__version__ = "0.1.0"` in `__init__.py`

Code

```toml
# pyproject.toml
[project]
name = "my_package"
dynamic = ["version"]

[tool.hatch.version]
source = "regex"
path = "src/my_package/__init__.py"
```

```python
# src/my_package/__init__.py
__version__ = "0.1.0"
```

So Let's Do Something About It

Approach #2:

- We can set the `version` field to dynamic…
- …and use the version control system (e. g., `git`) to determine the version for us
- We can then import the version from the file generated by `hatch-vcs`

Code

```
# pyproject.toml
[build-system]
requires = ["hatchling", "hatch-vcs"]
build-backend = "hatchling.build"

[project]
name = "my_package"
dynamic = ["version"]

[tool.hatch.version]
source = "vcs"

[tool.hatch.build.hooks.vcs]
version-file =
    "src/my_package/_version.py"


# src/my_package/__init__.py
from ._version import version

__version__ = version
```

File Selection

Hatch respects your `.gitignore` for what to include in each type of distribution:

SDist	Hatch will include everything ***not*** included in `.gitignore`, unless told otherwise.
Wheels	Everything in `src/<project>/` excluding files in your `.gitignore`.

Rewriting Paths

hatchling can also move files around and rewrite paths in your Distribution package:

```
[tool.hatch.build.targets.wheel]
include = ["src/my_package", "a-folder"]


[tool.hatch.build.targets.wheel.sources]
"src/my_package" = "my_package"
"a-folder" = "my_package/renamed_folder"
```

Data Files

Data Files Any files intended for use at ***runtime*** that are shipped with your package and are not code.

- → Can be configuration files or examples
- → Data files are best put in a well-defined directory that can be accessed by users

Data Files

- Set up your data files in your `pyproject.toml`:

```
[tool.hatch.build.targets.wheel.shared-data]
"a-file.json" = "share/a-file.json"
"a-directory" = "etc/a-directory"
```

- Access them, e. g., using `sysconfig` or `importlib`

```
# with sysconfig
import sysconfig

root = sysconfig.get_path("data", sysconfig.get_default_scheme())
file_path = root + "/share/a-file.json"


# with importlib_resources
from importlib_resources import files

file_path = files("my_package").joinpath("a-file")
```

Building and Inspecting a Wheel (Try It)

- A local package:

```
$ hatch build -t wheel:editable .
$ zipinfo ./my_package*.whl
```

- Or a package from PyPI:

```
$ pip wheel --quiet --no-deps pyvisgen
$ zipinfo ./pyvisgen*.whl
```

Further Reading: Packaging

- Packaging in Python (Angus Hollands)
- Python Packaging User Guide
- Python Packaging Authority
- hatch
- uv
- mamba
- Scientific Python Library Development Guide
- https://learn.scientific-python.org/development/patterns/data-files/

Code Quality

```python
def f(x,y=0):return[x[i]+y if x[i]>0else y-x[i]for i in range(len(x))]
```

Shamelessly taken from
Stefan’s talk :)

```
def f(x,y=0):return[x[i]+y if x[i]>0else y-x[i]for i in range(len(x))]
```

This runs, but do you trust it?

Shamelessly taken from Stefan's talk :)

Code Quality Terminology

1. Surface Quality

   → Formatting: Layout, naming conventions, whitespaces

2. Semantic Quality

   → Docstrings, type hinting

3. Testability

   → Writing (simple) code that is easy to test

Surface Quality: PEP 8

↗Python Enhancement Proposal No. 8 (PEP 8)

- Coding convention comprising the standard library, all about readability
- Key Aspects:

Code Layout **String Quotes** **Whitespaces**

Trailing Commas **Comments** **Naming Conventions**

Surface Quality: PEP 8

```python
def add(a, b): return a+b
from rich import print
import os, math
def printPi():print(math.pi)
```

→

```python
import math
import os

from rich import print


def add(a, b):
    return a + b


def print_pi():
    print(math.pi)
```

Surface Quality: Tools

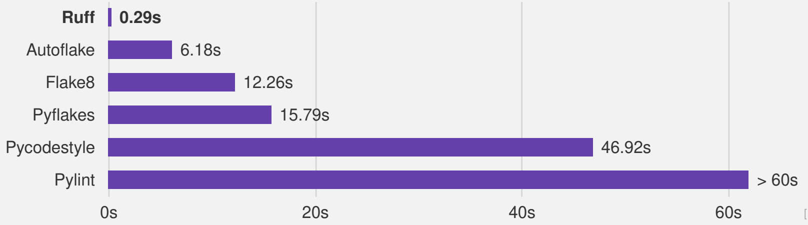
pycodestyle Strict PEP8 formatter.

flake8 `pyflakes` + `pycodestyle` + `optional plugins`.

black Follows PEP8 and adds it’s own strict rules (e. g., 88 chars limit).

isort Sorts imports so you don’t have to.

Ruff All of the above, highly customizable using `pyproject.toml` or `ruff.toml`, and extremely fast.



[docs.astral.sh/ruff]

Surface Quality: Ruff

- Install via `uv` as global tool or add to your project:

```
$ uv tool install ruff@latest
$ uv add --dev ruff
```

- Two tools in one: *Formatter* and *linter*

```
$ ruff check
$ ruff format
```

- Ruff supports over ***800*** lint rules, inspired by the popular tools shown earlier → See Rules .
  - → Configure everything in `pyproject.toml` or `ruff.toml`.
  - → Disable specific rules that you don’t need in your project, e. g., `B905` *zip-without-explicit-strict*.

pyproject.toml

```toml
[tool.ruff]
target-version = "py313"
line-length = 88
extend-exclude = ["tests"]

[tool.ruff.lint]
extend-select = [
  "I",   # isort
  "E",   # pycodestyle
  "F",   # Pyflakes
  "UP",  # pyupgrade
  "B",   # flake8-bugbear
  "SIM", # flake8-simplify
]
ignore = ["B905"]

fixable = ["ALL"]
unfixable = []

[tool.ruff.lint.per-file-ignores]
"examples/**" = ["I"]

[tool.ruff.format]
quote-style = "double"
indent-style = "space"
line-ending = "auto"
skip-magic-trailing-comma = false
docstring-code-format = true

[tool.ruff.lint.isort]
known-first-party = ["my_package"]
```

Semantic Quality: Docstrings

- Explains what your code does.

- Can be understood by IDEs and autocompletion tools

- Necessary for well-written docs (later)

- Structure:
  → Triple double quotes (`""" ... """`)
  - Human-readable, complete sentences describing your code
  - Explanation of parameters, returns, and exceptions

- Many different styles available: Use ***one*** and ***stick to it***.

NumPy Style

```python
def draw_sampling_opts(size: int) -> Dict:
    """Draws randomized sampling parameters
    for the simulation.

    Parameters
    ----------
    size : int
        Number of parameters to draw, equal
        to number of images.

    Returns
    -------
    samp_opts : dict
        Sampling options/parameters stored
        inside a dictionary.
    """
```

Semantic Quality: Type Hinting

Python is dynamically typed, but...

- ...you can still declare types for variables:

```
foo: int = 1
bar: str = "app"
baz: np.ndarray = np.array([ ... ])

def func(a: int, b: int=42) -> int:
  return a + b
```

→ Improved code readability

- IDE and linting support, e. g., through code completion
- But: Type hinting is ***not*** enforced at runtime and one has to consider dynamic types
- Tools:

↗ **mypy**	Good for CI/CLI
↗ **pyright**	Proprietary tool, but faster and with VSCode integration

Automation: pre-commit Hook

- pre-commit does all the formatting and linting for you

- Install via uv:

  `$ uv pip install pre-commit`

- Many different hooks available:
  - ruff, mypy, codespell, and many more...

- Runs all tools defined in `.pre-commit-config.yaml`

- Run `$ pre-commit install` to install hooks in your project

- pre-commit runs automatically whenever something is comitted using `$ git commit ...`

.pre-commit-config.yaml

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: "v5.0.0"  # <- git version tag
    hooks:
      - id: check-added-large-files
      - id: check-case-conflict
      - id: check-merge-conflict
      - id: check-symlinks
      - id: check-yaml
      - id: debug-statements
      - id: end-of-file-fixer
      - id: mixed-line-ending
      - id: name-tests-test
        args: ["--pytest-test-first"]
      - id: requirements-txt-fixer
      - id: trailing-whitespace

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: "v0.12.3"
    hooks:
      - id: ruff-format
      - id: ruff-check
        args: ["--fix", "--show-fixes"]

  - repo: https://github.com/codespell-project/codespell
    rev: v2.4.1
    hooks:
    - id: codespell
      additional_dependencies:
        - tomli
```

Further Reading: Code Quality

- Stefans Talk On Code Quality
- PEP8 – Style Guide for Python Code
- Ruff Docs
- mypy Docs
- pyright Docs
- pre-commit
- pre-commit-hooks
- codespell
- PEP484 – Type Hints
- PEP 544 – Protocols: Structural subtyping
- Scientific Python Library Development Guide: Type Checking
- NumPy Style Guide
- Google Python Style Guide

Testing

When Do We Need Tests?

Imagine the following...

- You have written a package with a lot of code, e. g., multiple scripts
- You found a bug somewhere in your code
- You have not thought of possible edge cases during development

→ You will need to investigate your codebase for causes of the bug and even then the same bug may appear some time later

Solution

Write persistent tests **during development!**

Solution

Write persistent tests **during development!**

(And **automate** them → see CI)

Test Levels

Unit Testing Test single units (i. e., single functions or classes) of your software.

Integration Testing Test multiple components that depend on each other.

System Testing Test the entire software with respect to its requirements, e. g., I/O data.

Operational Acceptance Testing Give your software to the user to break it.

Test Levels

Unit Testing Test single units (i. e., single functions or classes) of your software.

Integration Testing Test multiple components that depend on each other.

System Testing Test the entire software with respect to its requirements, e. g., I/O data.

Operational Acceptance Testing Give your software to the user to break it.

What Do We Test For?

This is probably the hardest part...

- You will need to understand your code
- You will need to verify how much and what parts of your code are covered by tests
- Even then your code may not be guaranteed to work error-free
- ***Good practice***: Every time you find a bug, add a unit test so it doesn't reappear

Tools

Shipped with Python:

doctest Allows you to write simple tests in the docstrings of your functions.

unittest Allows you to write regular unit tests, i. e., separate functions and classes that test your code.

Additional tools:

pytest Scalable, extensible (i. e., through plugins), and easy to use test framework.

Coverage.py A tool for measuring code coverage. Works well with `pytest` if `pytest-cov` is installed:

```
$ pytest --cov
```

tox A generic virtual environment management and test command line tool. Can be used to:

- → Check whether your package builds and installs in different envs
- → Run tests in each defined env, e. g., using `pytest`

Nox Similar to `tox`, but uses standard Python files and decorators for configuration. (For differences, see `Why I Like Nox` by Hynek Schlawack)

pytest

An example taken from the **pytest** docs

```python
# test_sample.py
def inc(x):
    return x + 1


def test_answer():
    assert inc(3) == 5
```

```
$ pytest test_sample.py
========================= test session starts ==========================
platform linux -- Python 3.x.y, pytest-8.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_sample.py F                                                  [100%]

=============================== FAILURES ===============================
_____________________________ test_answer ______________________________

    def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        + where 4 = inc(3)

test_sample.py:6: AssertionError
======================== short test summary info =======================
FAILED test_sample.py::test_answer - assert 4 == 5
=========================== 1 failed in 0.12s ==========================
```

pytest

- `pytest` runs all functions starting with `test_` (or classes starting with `Test`).
- Provide `pytest` with the file that contains the specific test functions you want to run.
- If no arguments are provided to `pytest`, it looks for paths defined in `testpath` (if defined)
  - → Otherwise: Recursive search for files matching `test_*.py` or `*_test.py`
- Prints are suppressed per default; use the `-s` flag to see prints:

```
$ pytest -s
```

pyproject.toml

```
[tool.pytest.ini_options]
testpaths = [
  "tests",
]
addopts = "--verbose"
```

Useful Feature: pytest Fixtures

```python
# contents of test_append.py (pytest)
import pytest


# Arrange
@pytest.fixture
def first_entry():
    return "a"


# Arrange
@pytest.fixture
def order(first_entry):
    return [first_entry]


def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]
```

```python
# radionets tests/conftest.py
import shutil

import pytest


@pytest.fixture(autouse=True, scope="session")
def test_suite_cleanup_thing():
    yield

    build = "./tests/build/"
    print("Cleaning up tests.")

    shutil.rmtree(build)
```

→ `pytest` fixtures provide defined, reliable and consistent context for the tests

→ Essentially code to be run before or after a test, e. g., to prepare objects, data, or files

Testing: Good Practices

Test-driven development:

- → Make testing part of your development process
- → Write tests ***before*** implementing your code:
  1. Specify what the code should do
  2. Write tests that test those specifications
  3. Implement the code

In reality this may not always be feasible, but…

- Always try to write tests for your code, especially for critical components
- You can always add tests at a later time, in a separate commit
- Always write tests when you found and fixed a bug to ensure it doesn't reappear

Further Reading: Testing

- Nikolai Krug's PYOPP Talk
- Nikolai Krug's pytest Tutorial
- doctest
- unittest
- pytest
- Coverage.py
- tox

- Nox and Why I Like Nox
- pytest-xdist
- pytest-regression
- pytest-mock
- pytest-hypothesis
- pytest-order
- Intro To Testing by Henry Schreiner

Documentation

Why Should We Document Our Code?

Well documented code improves...

- Maintainability: Future developers, debugging, ...
- Accessibility: Make your package easier to understand for new users
- Collaboration: Docs as a shared knowledge source

Tool Of Choice: Sphinx

- FOSS, extensible documentation generator written in Python
- Multiple output formats: **HTML**, LaTeX, ePub, and more...
- Content is written using a mark-up language (`reST` or `MyST`)
- Support for various docstring formats (some through extensions)
- Install via uv or mamba:

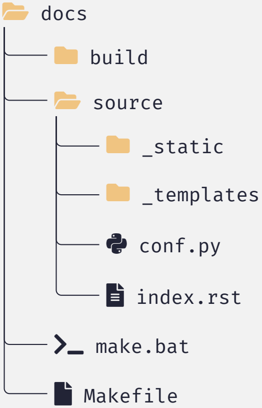
```
$ uv pip install sphinx
$ mamba install sphinx
```

Getting Started

```
$ sphinx-quickstart docs
> Separate source and build directories (y/n) [n]: y
> Project name: ...
> Author name(s): ...
> Project release []: ...
> Project language [en]: ...
```

Getting Started

```
$ sphinx-quickstart docs
> Separate source and build directories (y/n) [n]: y
> Project name: ...
> Author name(s): ...
> Project release []: ...
> Project language [en]: ...
```



docs

build

source

_static

_templates

conf.py

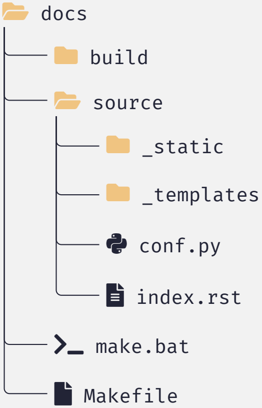
index.rst

make.bat

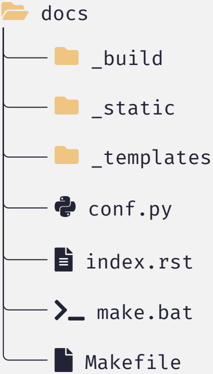
Makefile

Getting Started

```
$ sphinx-quickstart docs
> Separate source and build directories (y/n) [n]: y
> Project name:  ...
> Author name(s):  ...
> Project release []:  ...
> Project language [en]:  ...
```



- docs
  - build
  - source
    - _static
    - _templates
    - conf.py
    - index.rst
  - make.bat
  - Makefile



- docs
  - _build
  - _static
  - _templates
  - conf.py
  - index.rst
  - make.bat
  - Makefile

Breakdown of the Generated Structure

build: Output directory for the docs.

_static: Directory for static elements such as images, icons, or logos.

_templates: Used to store Jinja templates for HTML page generation.

index.rst: Root document; contains the root of the table of contents tree.

conf.py: Main configuration file written in Python.

Let's Build Our Docs

We will use the Makefile generated by sphinx-quickstart to build any format:

```
$ make <format>
```

So, for the HTML version:

```
$ make html
```

This will generate the HTML files for our docs inside the build directory. We can view the docs locally by running a Python HTTP server (in this case from inside the docs directory):

```
$ python -m http.server -d build/html [port]
```

Note

[port] is optional, see python -m http.server --help.

Setting Up conf.py

The `conf.py` file generated by Sphinx should look something like this:

```python
# -- Project information ------------------------
project = 'pyopp'
copyright = '2025, Author'
author = 'Author'
release = 'v0.1'

# -- General configuration ----------------------
extensions = []

templates_path = ['_templates']
exclude_patterns = []

# -- Options for HTML output --------------------
html_theme = 'alabaster'
html_static_path = ['_static']
```

Setting Up conf.py | Project Information

Let's get some metadata from `pyproject.toml` using `tomli` or `tomllib` (Python ≥ 3.11):

```python
#!/usr/bin/env python3
import datetime
import sys
from pathlib import Path

import package                                                           # your package

if sys.version_info < (3, 11):
    import tomli as tomllib
else:
    import tomllib

pyproject_path = Path(__file__).parent.parent.parent / "pyproject.toml"  # Get path of pyproject.toml
pyproject = tomllib.loads(pyproject_path.read_text())                    # Load contents

project = pyproject["project"]["name"]                                   # Get project name
author = pyproject["project"]["authors"][0]["name"]                      # Get author name
copyright = "{}.  Last updated {}".format(
    author, datetime.datetime.now().strftime("%d %b %Y %H:%M")
)                                                                        # Set copyright string
python_requires = pyproject["project"]["requires-python"]                # Get minimum python version requirement
rst_epilog = f"""
.. |python_requires| replace:: {python_requires}
"""                                                                      # Make python_requires var accessible

version = package.__version__                                            # Get version
release = version                                                        # Full release version
```

Setting Up conf.py | General Configuration

Sphinx extensions add functionality and customization. The following extensions are some of the extensions we always use in our docs:

```
extensions = [
    "sphinx.ext.autodoc",                # Imports modules and pulls in documentation from docstrings
    "sphinx.ext.intersphinx",            # Cross-references to other projects
    "sphinx.ext.coverage",               # Collects doc coverage stats
    "sphinx.ext.viewcode",               # Links to highlighted source code (i.e. "[source]" button)
    "sphinx_automodapi.automodapi",                    # Automatically generates module documentation
    "sphinx_automodapi.smart_resolver",                # Helps resolving some imports
    "numpydoc",                                        # Support for the NumPy docstring format
    "IPython.sphinxext.ipython_console_highlighting",  # Syntax highlighting of ipython prompts
    "sphinx_copybutton",                               # Adds a copybutton to code blocks
]
```

Setting Up conf.py | General Configuration

Some extensions are not shipped with Sphinx and need to be installed separately in your environment:

```
$ mamba install sphinx-automodapi numpydoc pydata-sphinx-theme sphinx-copybutton
```

or with uv

```
$ uv pip install sphinx-automodapi numpydoc pydata-sphinx-theme sphinx-copybutton
```

Setting Up conf.py | General Configuration

Now we can set up some more settings for the extensions:

```
# gets rid of some errors during build
numpydoc_show_class_members = False
numpydoc_class_members_toctree = False

intersphinx_mapping = {
    "numpy": ("https://numpy.org/doc/stable", None),
    ...
}

suppress_warnings = ["intersphinx.external"]  # sometimes necessary

templates_path = ["_templates"]
exclude_patterns = ["build", "Thumbs.db", ".DS_Store", "changes", "*.log"]

source_suffix = {".rst": "restructuredtext"}  # Set .rst files as source files for docs
master_doc = "index"                          # index.rst as root file
```

Setting Up conf.py | HTML And Theme Options

HTML options set the look of your docs. The Sphinx community has created a variety of themes you can choose from.

```
html_theme = "pydata_sphinx_theme"            # Modern, widely used theme

html_static_path = ["_static"]
html_favicon = "_static/favicon/favicon.ico"  # Icon file for browser tabs
html_css_files = ["custom.css"]               # Custom CSS settings like colors or fonts
html_file_suffix = ".html"

html_theme_options = { ... }                  # Depends on the theme

html_title = f"{project}"                     # e.g. your project name
htmlhelp_basename = project + " docs"
```

Check out *Sphinx Themes Gallery* for a curated list of available themes: sphinx-themes.org

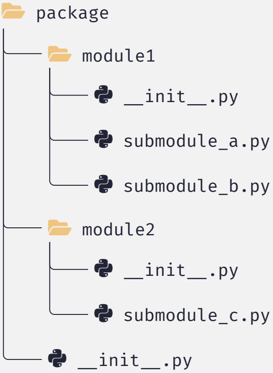
Filling the Docs: Landing Page

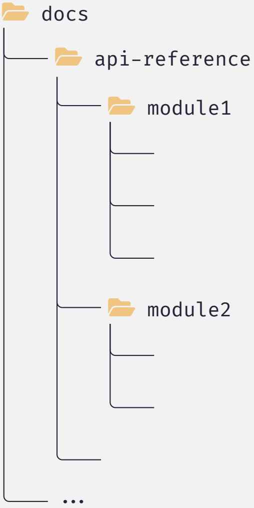
```rst
:html_theme.sidebar_secondary.remove: true
:html_theme.sidebar_primary.remove: true

.. _package:

=======
Package
=======

.. currentmodule:: package

**Version**: |version| | **Date**: |today|

**Useful links**: `Source Repository <https://github.com/your_project/package>`__ |
`Issue Tracker <https://github.com/your_project/package/issues>`__ |
`Pull Requests <https://github.com/your_project/package/pulls>`__

**License**: `MIT <https://github.com/your_project/package/blob/main/LICENSE>`__

**Python**: |python_requires|

.. toctree::
   :maxdepth: 1
   :hidden:

   api-reference/index
   changelog
```

Filling the Docs: API References

We will create the API references (semi-)automatically in a few steps:

1. Copy the structure of your actual package

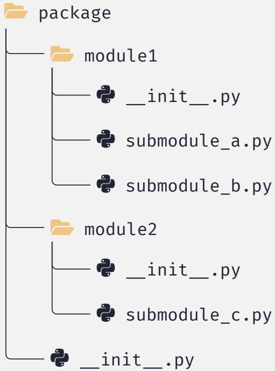


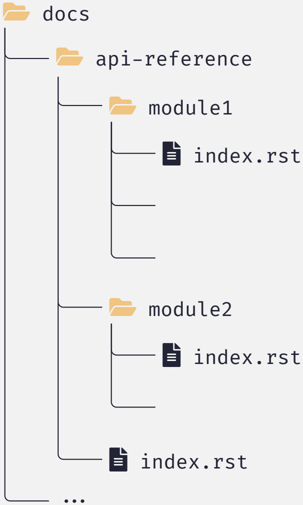


Filling the Docs: API References

We will create the API references (semi-)automatically in a few steps:

1. Copy the structure of your actual package
2. Populate every subdirectory with a `index.rst`

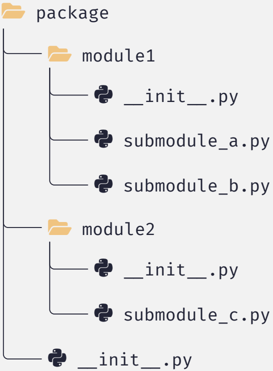




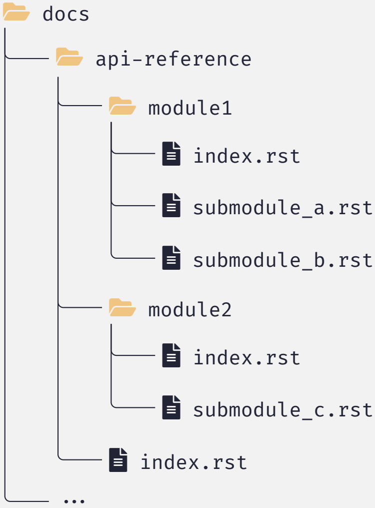
Filling the Docs: API References

We will create the API references (semi-)automatically in a few steps:

1. Copy the structure of your actual package
2. Populate every subdirectory with a `index.rst`
3. Create separate `.rst` files for every submodule



```
package
├── module1
│   ├── __init__.py
│   ├── submodule_a.py
│   └── submodule_b.py
├── module2
│   ├── __init__.py
│   └── submodule_c.py
└── __init__.py
```



```
docs
├── api-reference
│   ├── module1
│   │   ├── index.rst
│   │   ├── submodule_a.rst
│   │   └── submodule_b.rst
│   ├── module2
│   │   ├── index.rst
│   │   └── submodule_c.rst
│   └── index.rst
└── ...
```

Filling the Docs: API References

For now, the API reference will still be empty. We have to fill in the `index.rst` files to change that. Starting with `api-reference/index.rst`:

```
.. _api-reference:

*************
API Reference
*************

.. toctree::
  :maxdepth: 1
  :glob:

  */index
```

We add...

1. A tag `.. _api-reference:` to the file so we can reference it if necessary
2. A title, e. g., “API Reference”
3. The table of contents with the `.. toctree::` directive
    - And add only `index.rst` files from the subdirectories to the TOC

Filling the Docs: API References

```rst
.. _module1:

********************************
Module1 (:mod:`package.module1`)
********************************

.. currentmodule:: package.module1

Introduction
============

:mod:`package.module1` contains useful methods and classes.

Submodules
==========

.. toctree::
  :maxdepth: 1
  :glob:

  submodule_a
  submodule_b

Reference/API
=============

.. automodapi:: package.module1
    :no-inheritance-diagram:
```

Now, we do the same for the `index.rst` files in the module directories:

We add...

1. A tag and module title
2. The `.. currentmodule::` directive to let Sphinx know that classes and functions documented from here on are in the given module
3. (optional) Some introduction to the module
4. The table of contents for the submodules of the module
5. The `.. automodapi::` directive for the current module to get a list of classes and functions

Filling the Docs: API References

Finally, we write the submodule `.rst` files:

```rst
.. _submodule_a:

************************************************
submodule_a (:mod:`package.module1.submodule_a`)
************************************************

.. currentmodule:: package.module1.submodule_a

Submodule of :mod:`package.module1`.


Reference/API
=============

.. automodapi:: package.module1.submodule_a
    :inherited-members:
```

We add...

1. A tag, the submodule title, and the `.. currentmodule::` directive
2. (optional) Some introduction to the submodule
3. The `.. automodapi::` directive for the current submodule to get a list of classes and functions

reST: Headings

```
####
Part
####

*******
Chapter
*******

Section
=======

Subsection
----------

Subsubsection
^^^^^^^^^^^^^

Paragraph
"""""""""
```

- The structure is technically determined by order of occurance
  - **But**: For better readability stick to the same order throughout your docs, e. g., the one shown here (recommended)

- While overlines are optional, they are encouraged for parts and chapters

- Any of the following symbols are valid for over- and underlines:
  `# * = - ^ " + _ ~` ` . , : ; ' ! ? & $ % ( ) [ ] { } < > @ \ / |

reST: Roles, Directives, and Field Lists

- Roles are **inline** pieces of explicit markup that are understood by Sphinx. The syntax is:

```
:rolename:`content`
```

→ Examples:

```
:mod:`package.module1` :code:`foo = 42` :math:`F = m\cdot a`
```

- Directives are **blocks** of explicit markup that are understood by Sphinx. The syntax is:

```
.. directive:: [(optional) arguments]
   [:(optional) field list:] [(optional) field list value]

   [Body elements of the directive]
```

→ Examples:

```
.. image:: picture.png
   :width: 90%
   :alt: A nice picture.

.. code-block::
   :caption: A code block.

   def func(param: int) -> int: ...
```

Hosting on ReadtheDocs

- Free, if your package is open-source, i. e., publically available on, e. g., GitHub or GitLab and no handling of secrets required
- Allows you to preview your docs on every PR
- Works seamlessly with Sphinx
- Automatically builds the docs from your `main` branch
- Supports downloading the docs in PDF or other formats

Hosting on ReadtheDocs

1. Set up a `.readthedocs.yaml` file in your repository:

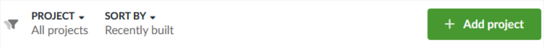
```yaml
version: 2

build:
  os: ubuntu-24.04
  apt_packages:
    - graphviz
  tools:
    python: "3.13"
  jobs:
    pre_create_environment:
        - asdf plugin add uv
        - asdf install uv latest
        - asdf global uv latest
    install:
      - uv pip install --upgrade pip  # <- may be necessary if pip < 25
      - uv pip install --group docs .

sphinx:
  configuration: docs/conf.py
```

Hosting on ReadtheDocs

2. Sign up/log in to **Read*the*Docs** (Community), e. g., via GitHub, GitLab, or Bitbucket

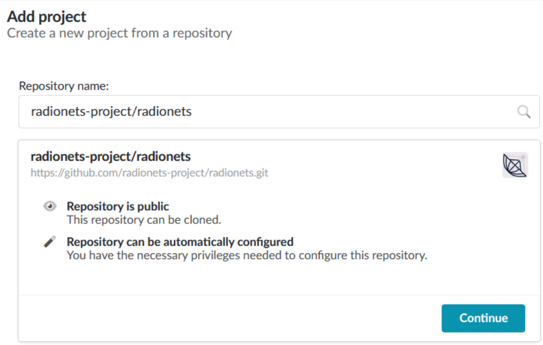
Hosting on ReadtheDocs

2. Sign up/log in to ReadtheDocs (Community), e. g., via GitHub, GitLab, or Bitbucket
3. In your dashboard, click on “Add project”



Hosting on ReadtheDocs

2. Sign up/log in to ReadtheDocs (Community), e. g., via GitHub, GitLab, or Bitbucket
3. In your dashboard, click on “Add project”
4. Search for your repository and click “Continue”



Add project

Create a new project from a repository

Repository name:

radionets-project/radionets

radionets-project/radionets

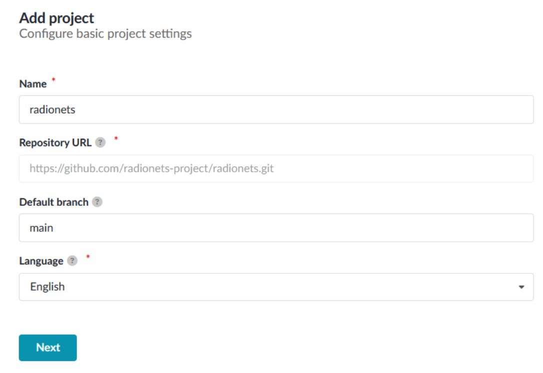
https://github.com/radionets-project/radionets.git

Repository is public
This repository can be cloned.

Repository can be automatically configured
You have the necessary privileges needed to configure this repository.

Hosting on ReadtheDocs

2. Sign up/log in to ReadtheDocs (Community), e. g., via GitHub, GitLab, or Bitbucket
3. In your dashboard, click on “Add project”
4. Search for your repository and click “Continue”
5. Configure the basic settings and click “Next”



Add project

Configure basic project settings

Name *

radionets

Repository URL ? *

https://github.com/radionets-project/radionets.git

Default branch ?

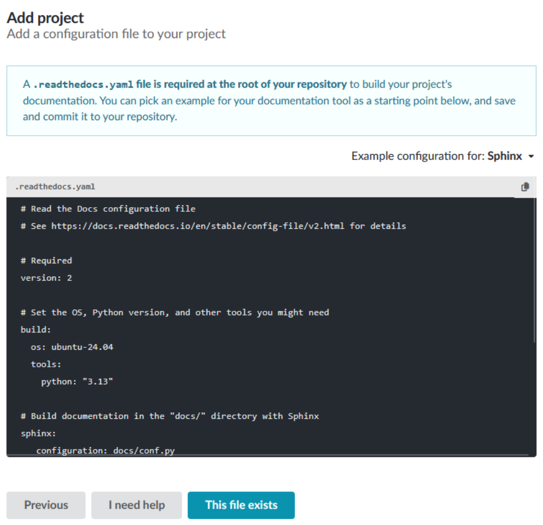
main

Language ? *

English

Hosting on ReadtheDocs

2. Sign up/log in to ReadtheDocs (Community), e. g., via GitHub, GitLab, or Bitbucket
3. In your dashboard, click on “Add project”
4. Search for your repository and click “Continue”
5. Configure the basic settings and click “Next”
6. Ensure the `.readthedocs.yaml` file exists in your repository, and click “This file exists”



Add project

Add a configuration file to your project

A **`.readthedocs.yaml` file is required at the root of your repository** to build your project's documentation. You can pick an example for your documentation tool as a starting point below, and save and commit it to your repository.

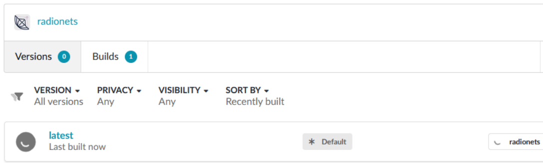
Example configuration for: **Sphinx**

.readthedocs.yaml

```
# Read the Docs configuration file
# See https://docs.readthedocs.io/en/stable/config-file/v2.html for details

# Required
version: 2

# Set the OS, Python version, and other tools you might need
build:
  os: ubuntu-24.04
  tools:
    python: "3.13"

# Build documentation in the "docs/" directory with Sphinx
sphinx:
   configuration: docs/conf.py
```

Hosting on ReadtheDocs

2. Sign up/log in to ReadtheDocs (Community), e. g., via GitHub, GitLab, or Bitbucket
3. In your dashboard, click on “Add project”
4. Search for your repository and click “Continue”
5. Configure the basic settings and click “Next”
6. Ensure the `.readthedocs.yaml` file exists in your repository, and click “This file exists”
7. Your docs should now be building and will be rebuilt anytime a PR is merged into `main`



radionets

latest
Last built now

Further Reading: Docs

- My PYOPP Talk
- Sphinx
- sphinx-autobuild
- Import System
- PEP 420 – Implicit Namespace Packages
- reStructuredText (reST)
- Roles
- Directives
- Field Lists
- Towncrier (Changelogs)
- sphinx-automodapi
- PyData Sphinx Theme
- numpydoc
- sphinx-design
- sphinx-gallery

Continuous Integration (CI), Deployment, and Continuous Delivery (CD)

What is Continuous Integration?

- A practice where tests and builds are run automatically, e. g., after code changes were merged/committed
- Goal: Find bugs, improve software quality (e. g., performance) and ensure your software runs on different platforms
- Every commit triggers a CI job
- Addressing failed CI jobs before merging a PR ensures code quality
- Running tests locally before committing adds an extra layer of ensuring code quality

Note

The quality of your CI strongly depends on the quality of your tests.

→ Requires effort beforehand.

CI: Multiple Platforms | GitHub Actions

```yaml
name: CI

on:
  push:
    branches:
      - main
    tags:
      - '**'
  pull_request:

env:
  MPLBACKEND: Agg
  PYTEST_ADDOPTS: --color=yes
```

CI: Multiple Platforms | GitHub Actions

2. We will be using GitHub Actions' matrix strategy to define multiple platforms:

```yaml
jobs:
  tests:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        include:
          - os: ubuntu-latest
            python-version: "3.10"
            install-method: mamba

          - os: ubuntu-latest
            python-version: "3.12"
            install-method: mamba
            extra-args: ["codecov"]  # lead platform for code cov

          - os: ubuntu-latest
            python-version: "3.12"
            install-method: pip

          - os: macos-13
            python-version: "3.10"
            install-method: pip

    defaults:
      run:
        # We need login shells (-l) for micromamba to work.
        shell: bash -leo pipefail {0}
```

CI: Multiple Platforms | GitHub Actions

3. Adding steps:

```yaml
steps:
  - uses: actions/checkout@v4
    with:
      fetch-depth: 0

  - name: Prepare mamba installation
    if: matrix.install-method == 'mamba' &&  contains(github.event.pull_request.labels.*.name, 'documentation-only') == false
    env:
      PYTHON_VERSION: ${{ matrix.python-version }}
    run: |
      # setup correct python version
      sed -i -e "s/- python=.*/- python=$PYTHON_VERSION/g" environment.yml

  - name: mamba setup
    if: matrix.install-method == 'mamba' && contains(github.event.pull_request.labels.*.name, 'documentation-only') == false
    uses: mamba-org/setup-micromamba@v1
    with:
      environment-file: environment.yml
      cache-downloads: true

  - name: Python setup
    if: matrix.install-method == 'pip' && contains(github.event.pull_request.labels.*.name, 'documentation-only') == false
    uses: actions/setup-python@v5
    with:
      python-version: ${{ matrix.python-version }}
      check-latest: true
```

CI: Multiple Platforms | GitHub Actions

4. For macOS, we have to fix the Python path:

```yaml
steps:
  - ...

  - if: matrix.install-method == 'pip' && runner.os == 'macOS' && contains(github.event.pull_request.labels.*.name,
  ↪  'documentation-only') == false
    name: Fix Python PATH on macOS
    run: |
      tee -a ~/.bash_profile <<<'export PATH="$pythonLocation/bin:$PATH"'
```

Multiple Platforms | GitHub Actions

5. Install dependencies and run tests:

```yaml
steps:
  - ...

  - uses: astral-sh/setup-uv@v6

  - name: Install dependencies
    env:
      PYTHON_VERSION: ${{ matrix.python-version }}
    run: |
      python --version
      uv pip install --group tests -e .
      uv pip freeze
      uv pip list

  - name: List installed package versions (conda)
    if: matrix.environment-type == 'mamba'
    run: micromamba list

  - name: Tests
    run: |
      pytest -vv --cov --cov-report=xml

  - name: Upload coverage to Codecov
    uses: codecov/codecov-action@v4
    env:
      CODECOV_TOKEN: ${{ secrets.CODECOV_TOKEN }}  # make sure you have this set as repository secret
```

Codecov

1. Sign up/log in to Codecov, e. g., via GitHub, GitLab, or Bitbucket
2. Select your repository from your dashoard
3. Select a setup option, e. g., “Using GitHub Actions”
4. Select an upload token. For a single repository, the repository token is sufficient
5. Add the token as repository secret
6. Update your CI to automatically upload the coverage to Codecov (after the `Tests` step of your job)

```yaml
- name: Tests
  run: |
    pytest -vv --cov --cov-report=xml

- name: Upload coverage to Codecov
  uses: codecov/codecov-action@v4
  env:
    CODECOV_TOKEN: ${{ secrets.CODECOV_TOKEN }}
```

→ **NEVER** share your token with anyone.

Linting With the CI

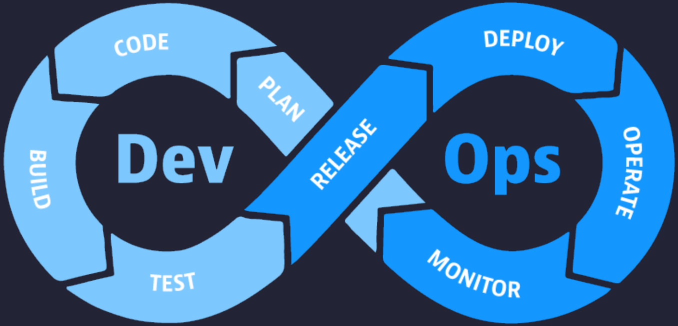
This is easier than ever: Set up your `.pre-commit-config.yaml`, then go to `pre-commit.ci` and add your project/repository.

Building the Docs With the CI

The docs job can be started last.

```yaml
jobs:
  docs:
      runs-on: ubuntu-24.04
      steps:
        - uses: actions/checkout@v4
          with:
            fetch-depth: 0

        - name: Set up Python
          uses: actions/setup-python@v5
          with:
            python-version: "3.12"

        - name: Install doc dependencies
          run: |
            sudo apt update -y && sudo apt install -y git build-essential pandoc graphviz ffmpeg
            pip install -U pip towncrier setuptools
            pip install -e .[docs]
            git describe --tags

        - name: Build docs
          run: make -C docs html
```

CI/CD: DevOps


CODE
PLAN
BUILD
Dev
TEST
RELEASE
DEPLOY
Ops
OPERATE
MONITOR

CD: Publish on PyPI

```yaml
name: Build Python Package

on:
  push:
  workflow_dispatch:
  release:
    types:
      - published

jobs:
  dist:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: hynek/build-and-inspect-python-package@v2
        with:
          path: .
```

CD: Publish on PyPI (cont.)

```yaml
jobs:
  distlong:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - uses: astral-sh/setup-uv@v6

      - name: Build SDist and wheel
        run: uvx --from build pyproject-build

      - uses: actions/upload-artifact@v4
        with:
          name: Packages-distlong-${{ github.job }}
          path: dist/*

      - name: Check metadata
        run: uvx twine check ./dist/*
```

CD: Publish on PyPI (cont.)

```yaml
jobs:
  publishtrusted:
    needs: [ dist ]
    environment: pypi
    permissions:
      id-token: write
      attestations: write
      contents: read
    runs-on: ubuntu-latest
    if: github.event_name == 'release' && github.event.action == 'published'
    steps:
      - uses: actions/download-artifact@v4
        with:
          name: Packages
          path: dist

      - name: Generate artifact attestation for sdist and wheel
        uses: actions/attest-build-provenance@v2
        with:
          subject-path: "./dist/*"

      - uses: pypa/gh-action-pypi-publish@release/v1
```

Further Reading: CI/CD

- Jonas Eschle's PYOPP Talk
- My PYOPP Talk
- GitHub Actions
- actions/checkout
- astral-sh/setup-uv
- setup-micromamba

- Codecov
- Matrix Strategies
- pre-commit.ci
- GitLab CI
- GitLab: Predefined Variables
- badge.fury.io and shields.io (Badges)

MY COWORKERS WATCHING ME DEPLOY A "SMALL FIX" ON A FRIDAY
GET READY, EVERYBODY. HE'S ABOUT TO DO SOMETHING STUPID.