Documentation and Continuous Integration (CI)

Anno Knierim

June 25, 2025

Documentation

What You Will Learn:

- Documenting your code using *Sphinx*
- *reStructuredText* (`reST`/RST) syntax
- Using **Read***the***Docs**

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

Documentation

Sphinx

What is Sphinx?

- Open-source, extensible documentation generator written in Python
- Multiple output formats: HTML, LaTeX (for PDF), ePub, and more...
- Creates cross-references within your project and across different projects
- Allows documentation using a mark-up language (reST)
- Supports various docstring formats (some through extensions)

There are also alternatives to Sphinx, like *MkDocs* and *pdoc*, but Sphinx can be considered the industry standard for Python docs.

Installation

Sphinx can be installed via standard package managers:

- Installing from PyPI using `pip`:

```
$ pip install -U sphinx
```

- Conda/Mamba:

```
$ mamba install sphinx
$ conda install -c conda-forge sphinx
```

- Debian/Ubuntu using `apt`:

```
# apt install python3-sphinx
```

- Fedora Linux, RHEL, CentOS using `yum` or `dnf`:

```
# yum install python-sphinx
# dnf install python-sphinx
```

- Homebrew:

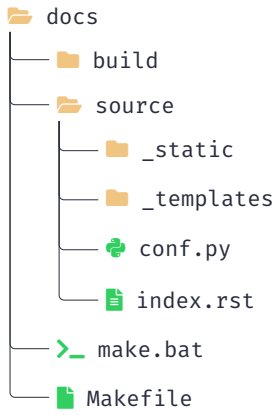
```
$ brew install sphinx-doc
```

Getting Started

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

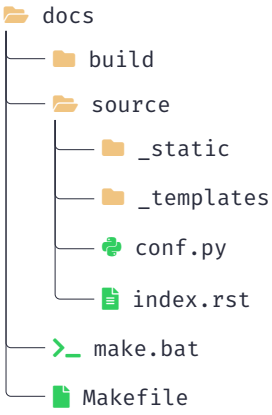
Getting Started

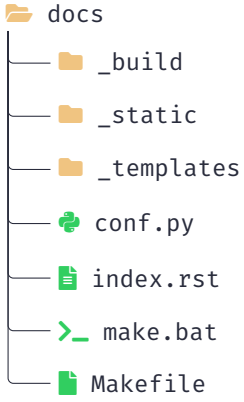
```
$ sphinx-quickstart docs
> Separate source and build directories (y/n) [n]: y
> Project name: ...
> Author name(s): ...
> Project release []: ...
> Project language [en]: ...
```



Getting Started

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





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`.

Alternative: sphinx-autobuild

- `sphinx-autobuild` rebuilds your documentation anytime it detects changes in your `docs/` directory
- Install it via pip:

  ```
  $ pip install sphinx-autobuild
  ```

- To build the docs, run:

  ```
  $ sphinx-autobuild docs docs/build/html
  ```

  This will start a server at `http://127.0.0.1:8000/`:

  ```
  [sphinx-autobuild] Starting initial build
  [sphinx-autobuild] > python -m sphinx build docs docs/build/html
  ...
  [sphinx-autobuild] Serving on http://127.0.0.1:8000
  [sphinx-autobuild] Waiting to detect changes...
  ```

Setting Up `conf.py`

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

Code | `docs/conf.py`

```
# -- 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

When using a **`pyproject.toml`** file for our project, we automatically get the metadata from that file using **`tomli`** or **`tomllib`** (Python $\geqslant$ **`3.11`**):

Code | `docs/conf.py`

```
#!/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"
pyproject = tomllib.loads(pyproject_path.read_text())

project = pyproject["project"]["name"]
author = pyproject["project"]["authors"][0]["name"]
copyright = "{}.  Last updated {}".format(
    author, datetime.datetime.now().strftime("%d %b %Y %H:%M")
)
python_requires = pyproject["project"]["requires-python"]
rst_epilog = f"""
.. |python_requires| replace:: {python_requires}
"""

version = pyvisgen.__version__
release = version  # full release version
```

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

Code | docs/conf.py

```
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
]
```

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

Code | docs/conf.py

```
# 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"  # Set .rst files as source files for docs
master_doc = "index"    # index.rst as root file
```

Setting Up `conf.py` | General Configuration

Some extensions are external and need to be installed separately in your environment:

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

or with `pip`

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

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.

Code | `docs/conf.py`

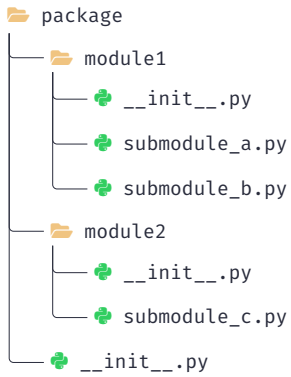
```
html_theme = "pydata_sphinx_theme"            # Modern, widely used theme
html_static_path = ["_static"]

html_file_suffix = ".html"

html_css_files = ["custom.css"]               # Custom CSS settings like colors or fonts

html_favicon = "_static/favicon/favicon.ico"  # Icon file for browser tabs

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

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

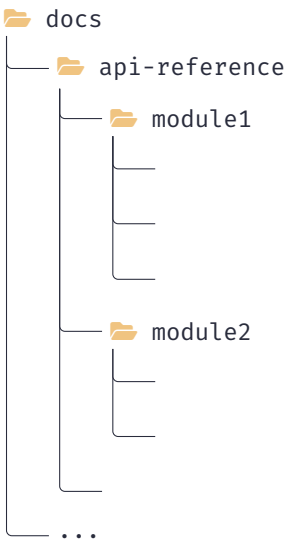
Check out *Sphinx Themes Gallery* for a curated list of available themes: sphinx-themes.org

Filling the Docs With Some API References

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

1. Copy the structure of your actual package

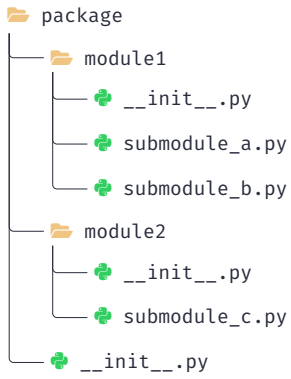


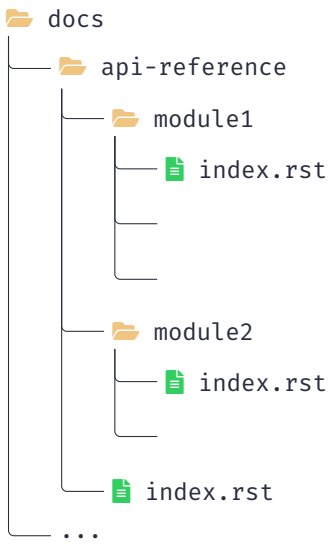


Filling the Docs With Some API References

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`

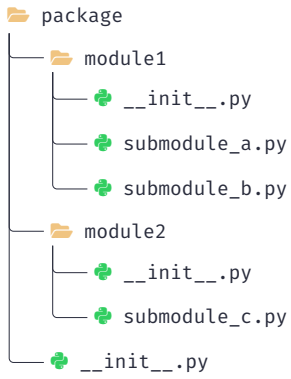


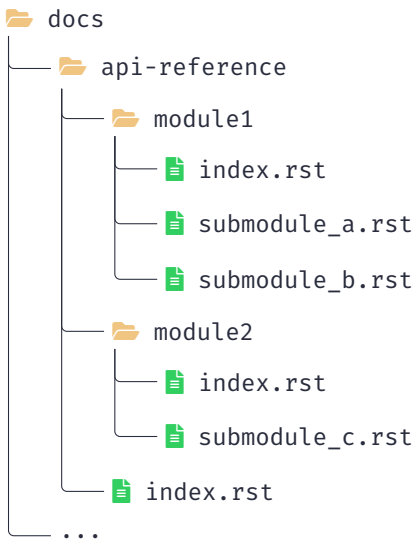


Filling the Docs With Some API References

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





`__init__.py`

Docs: Import System | Docs: PEP420

- `__init__.py` create structure in your package
  - This creates modules in your package
- Packages without `__init__.py` are *namespace* packages
- Sphinx (automodapi) requires the module structure to understand your package

Filling the Docs With Some 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`:

Code | `docs/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 With Some API References

Code | docs/api-reference/[module]/index.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 With Some API References

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

Code | `docs/api-reference/[module]/[submodule].rst`

```
.. _data:

**********************************************
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

Creating a Nice Landing Page and Adding the API Reference

Code | docs/index.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
```

Documentation

reStructuredText (`reST`)

reST Basics

- Paragraphs are the fundamental text blocks in reST, i.e., text chunks at the same indentation and separated by blank lines
- Inline Markup:

Code

```
*text* **text** ``text``
```

Output

text **text** `text`

- Hyperlinks:

Code

```
This text contains `a link`_.

.. _a link: https://erumdatahub.de


This text contains an
embedded `link <https://erumdatahub.de>`__.
```

Output

This text contains a link.

This text contains an embedded link.

Tables

- Basic tables are similar to Markdown tables
- Rendering depends on your theme

Code

```
+------------------+------------+----------+
| Header 1         | Header 2   | Header 3 |
+==================+============+==========+
| Column 1, Row 1  | 1.00       | 42       |
+------------------+------------+----------+
| Column 1, Row 2  | ...        | ...      |
+------------------+------------+----------+
```

CSV Tables

```
.. csv-table:: Table Title
   :header: "Header 1", "Header 2", "Header 3"
   :widths: 15, 10, 10

   "Column 1, Row 1", 1.00, 42
   "Column 1, Row 2", ..., ...
```

Lists

Code

```
* A bulleted list

- This is also a bulleted list.
- But this one has two items, and this
  item has two lines

1. A numbered list.

#. Autonumbering is also possible.
#. This is done using a ``#`` sign.
```

Nested Lists

```
* A bulleted list.

  * With a nested bulleted list.
  * Nested lists have to be separated by a blank line

* This continues the parent list.
```

Headings

Code

```
####
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:
  `# * = - ^ " + _ ~ ` . , : ; ' ! ? & $ % ( ) [ ] { } < > @ \ / |`

Roles

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 and Field Lists

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

```
.. directive:: [(optional) elements depending on directive]
   [:(optional) field list:]

   [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: ...
```

Documentation

Docstrings

Docstrings

Most of the documentation work will require you to write docstrings. The three most common formats are:

- `reST`
- Google
- `numpydoc`

Structure

```
"""[Summary of your method]

:param [Parameter name]: [Parameter description]
:type [Parameter name]: [Parameter type](, optional)
:returns: [Return description]
:rtype: [Return type]
:raises [Exception class]: [Exception description]
"""
```

Example

```
"""This is a reST-style docstring.

:param param1: First parameter.
:type param1: float
:param param2: Second parameter, defaults to None.
:type param2: str, optional
:returns: Some return value.
:rtype: int
:raises ValueError: Raises an exception.
"""
```

Structure

```
"""[Summary of your method]

Args:
    [Parameter name] ([Parameter type](, optional)): [Parameter description]

Returns:
    [Return type]: [Return description]

Raises:
    [Exception class]: [Exception description]
"""
```

Example

```
"""This is a Google-style docstring.

Args:
    param1 (float): First parameter.
    param2 (:obj:`str`, optional): Second parameter. Defaults to None.

Returns:
    int: Some return value.

Raises:
    ValueError: Raises an exception.
"""
```

Docs: numpydoc

Structure

```
"""[Summary of your method]

Parameters
----------
[Parameter name] : [Parameter type](, optional)
    [Parameter description]

Returns
-------
[Return name or type]( : [Return type if name was given])
    [Return description]

Raises
------
[Exception class]
    [Exception description]
"""
```

Example

```
"""This is a numpydoc-style docstring.

Parameters
----------
param1 : float
    First parameter.
param2 : str, optional
    Second parameter. Default: None

Returns
-------
int
    Some return value.

Raises
------
ValueError
    Raises an exception.
"""
```

Type Hinting

Docs: typing

→ Type hinting is the practice of declaring types for variables using a colon `:` after the variable name:

Code

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

→ Usually, type hinting is only applied to function definitions:

Code

```
def func(param1: int, param2: int=42) -> int:
    res = param1 + param2
    return res
```

→ Type hinting offers...
  → Improved code readability
  → IDE and linting support, e.g., through code completion

But: It is **not** enforced at runtime and one has to consider dynamic types.

Documentation

Changelogs

Why Track Changes and Use Changelogs?

- Changelogs are curated, chronogical lists of **notable** changes
- Users and (future) contributors of your package will be able to see what changes have been made for each release
- Changes should be grouped by type for better readability
- Every new version of your package should have an entry

```
Package v0.2.0 (2025-06-19)
===========================


API Changes
-----------


Bug Fixes
---------

  - Fixed a bug in ``submodule_a``
  - Fixed another bug in ``submodule_b``
    [`#1
     ↪  <https://github.com/your_project/package/pull/1>`__]


New Features
------------

  - Added ``module2``


Maintenance
-----------

  - Deleted unused code


Refactoring and Optimization
----------------------------

  - Refactored parts of ``submodule_a``
```

Create Changelogs With `towncrier`

Docs: `towncrier`

- `towncrier` is a tool to create changelogs for your project
- Instead of adding every little change, you will create “news fragments” that contain only **notable** changes
- `towncrier` will read these fragments and help you create a changelog for each release of your package
- Install `towncrier` via pip or mamba/conda:

```
$ pip install towncrier
$ mamba install towncrier
```

Setting Up `towncrier`

- `towncrier` is configured using your `pyproject.toml` or a `towncrier.toml` file
- A basic configuration is telling `towncrier` where to find news fragments and where to create the changelog:

Code

```
[tool.towncrier]
    package = "package"
    directory = "docs/changes"
    filename = "CHANGES.rst"
```

- We can also make use of a template and issue format: (Good example for a template)

Code

```
[tool.towncrier]
    ...
    template = "docs/changes/template.rst"
    # let towncrier create proper links to the merged PR
    issue_format = "`#{issue}
     ↪  <https://github.com/your_project/package/pull/{issue}>`__"
```

Setting Up towncrier

Code

```
[tool.towncrier]
    ...
    [tool.towncrier.fragment.feature]
        name = "New Features"
        showcontent = true

    [tool.towncrier.fragment.bugfix]
        name = "Bug Fixes"
        showcontent = true

    [tool.towncrier.fragment.api]
        name = "API Changes"
        showcontent = true

    [tool.towncrier.fragment.optimization]
        name = "Refactoring and Optimization"
        showcontent = true

    [tool.towncrier.fragment.maintenance]
        name = "Maintenance"
        showcontent = true

    [[tool.towncrier.section]]
        name = ""
        path = ""
```

- You can also set up the fragment types you are using
- Fragments are saved under the `directory` path set in the config, e.g., `docs/changes/`
- The syntax for the fragment names is `[PR Number].[fragment type].rst`

  Examples:

  `1.feature.rst`
  `1.bugfix.rst`
  `2.maintenance.rst`
- Multiple fragments can have the same leading PR number if they belong to the same PR

Building the Changelog and Adding It to the Docs

- To build the changelog, `towncrier` provides a CLI tool:
  ```
  $ towncrier build
  ```
- This creates the `CHANGES.rst` file set in the config, or prepends to it if it already exists
- `towncrier` will ask you whether it can delete the used news fragments automatically
- To include the changelog in the docs, you can create a file `docs/changelog.rst`:

Code

```
**********
Change Log
**********

.. include:: ../CHANGES.rst
```

Documentation

Hosting and Publishing

Hosts For Your Docs

So far, the docs are either in your local repository or also pushed to the remote, but not published. There are several ways to host your docs to make them available for your community:

- → **Read***the***Docs**
- → GitHub pages
- → Other online hosting services

The most common host is **Read***the***Docs**, which will be introduced in the following.

- Free, if your package is open-source, i.e., publically available on, e.g., GitHub or GitLab
- No secret handling 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 supported by ethical ads

Getting Started With **Read*the*Docs**

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

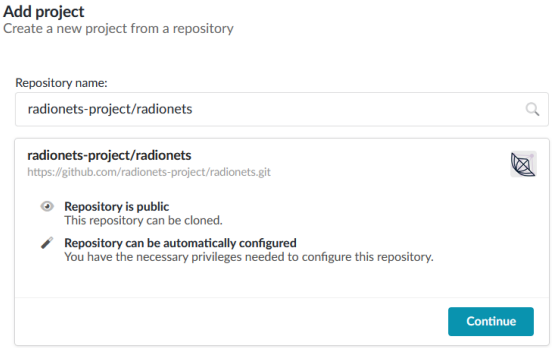
Code

```yaml
version: 2

build:
  os: ubuntu-24.04
  apt_packages:
    - graphviz
  tools:
    python: "3.13"  # or whatever Python version you prefer

python:
  install:
    - method: pip
      path: .
      extra_requirements:
        - docs

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

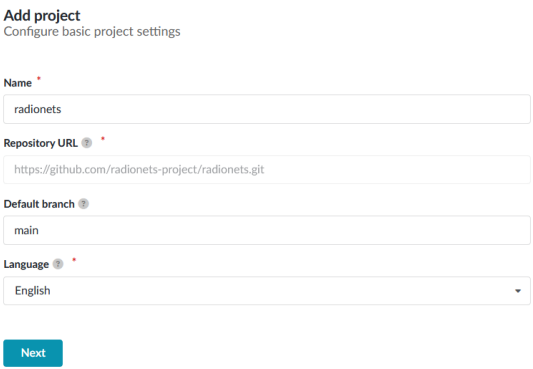
2. Sign up/log in to **Read*the*Docs** (Community), e.g., via GitHub, GitLab, or Bitbucket

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

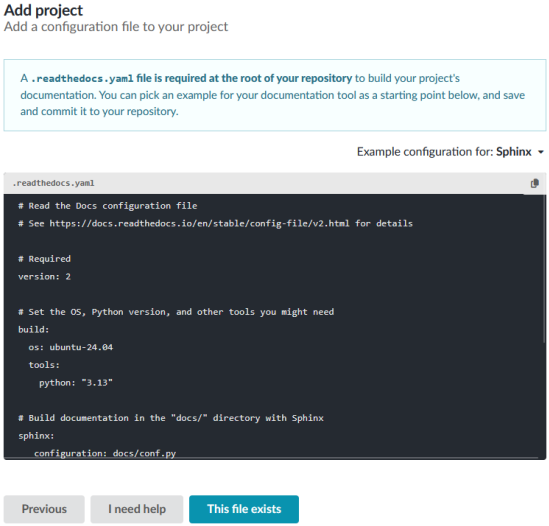
2. Sign up/log in to **Read*the*Docs** (Community), e.g., via GitHub, GitLab, or Bitbucket
3. In your dashboard, click on “Add project”
4. Search for your repository and click “Continue”



2. Sign up/log in to **Read*the*Docs** (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”

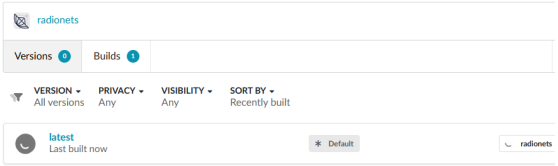


2. Sign up/log in to **Read*the*Docs** (Community), e.g., via GitHub, GitLab, or Bitbucket
3. In your dashboard, click on “Add project”
4. Search for your repository and click “Continue”
5. Configure the basic settings and click “Next”
6. Ensure the `.readthedocs.yaml` file exists in your repository,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



Getting Started With **Read*the*Docs**

2. Sign up/log in to **Read*the*Docs** (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`



Continuous Integration (CI)

What You Will Learn:

- Using CIs to test your code on GitHub or GitLab for multiple platforms
- Code coverage
- Linting using your CI
- Adding badges to your repository

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 Services

There are many CI services to choose from. Three widely used services are:

Jenkins Self-hosted, open-source CI service. One of the oldest CI services, going back as far as 2005 when it was called Hudson.

GitHub Actions A modular CI service developed by Microsoft. Multiple OS support and easy to maintain.

GitLab CI Widely used in GitLab repos, but also works with other services. Can be self-hosted or hosted centrally by GitLab.

→ We will focus on GitHub Actions (GHA) and GitLab CI in the following.

Continuous Integration (CI)

Getting Started | GitHub Actions

Docs: GitHub Actions

1. Create a `ci.yml` file in the `.github/workflows` directory (create the directory if necessary)
2. Set up some basics in the CI file:

Code

```yaml
name: CI

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

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

Docs: GitHub Actions

1. Create a `ci.yml` file in the `.github/workflows` directory (create the directory if necessary)
2. Set up some basics in the CI file:

Code

```yaml
name: CI

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

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

2.1 Name the CI, especially if you are running multiple CIs/CDs

Docs: GitHub Actions

1. Create a `ci.yml` file in the `.github/workflows` directory (create the directory if necessary)
2. Set up some basics in the CI file:

Code

```yaml
name: CI

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

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

2.1 Name the CI, especially if you are running multiple CIs/CDs

2.2 Set up when the CI should be run, e.g., on every push/merge to `main` and for every PR

Docs: GitHub Actions

1. Create a `ci.yml` file in the `.github/workflows` directory (create the directory if necessary)
2. Set up some basics in the CI file:

Code

```yaml
name: CI

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

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

2.1 Name the CI, especially if you are running multiple CIs/CDs

2.2 Set up when the CI should be run, e.g., on every push/merge to `main` and for every PR

2.3 Set up some environment variables, such as the matplotlib backend and color output for pytest

3. Set up a job for your CI that tests your code:

Code

```yaml
jobs:
  tests:  # Name of the job
    runs-on: ubuntu-latest

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

See About login shells... why login shells are necessary here.

Docs: `actions/checkout` Docs: `setup-micromamba`

3. Set up a job for your CI that tests your code (cont.):

Code

```yaml
tests:
  ...

  steps:
    - uses: actions/checkout@v4

    - uses: mamba-org/setup-micromamba@v1
      with:
        environment-file: environment.yml

    - name: Install dependencies
      run: |
        python --version
        pip install -e .[tests]
        pip freeze
        pip list

    - name: List installed package versions (conda)
      run: micromamba list

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

Continuous Integration (CI)

Code Coverage

Code Coverage

Code coverage shows you how much of your code is covered by tests.

- → While the reports produced by pytest in the CI are nice, more elaborate reports are sometimes desireable
- → Code review services provide such reports
- → Three commonly used services that are free for FOSS and support multiple languages are:
  - → Codecov
  - → SonarQube
  - → Codacy

We will focus on Codecov here, because of its simplicity.

Docs: 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)

Code

```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 }}
```

Important

NEVER share your token with anyone.

Continuous Integration (CI)

Multiple Platforms

Multiple Platforms | GitHub Actions

- So far we tested only on one platform
- We have to ensure our package also runs on all target platforms and all target versions of Python

We will modify our CI to test on multiple platforms:

1. The first part remains the same:

Code

```yaml
name: CI

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

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

Docs: Matrix Strategies

2. This time, we will be using GitHub Actions' matrix strategy to define multiple platforms:

Code

```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}
```

3. Adding steps:

Code

```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
```

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

Code

```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"'
```

5. Install dependencies and run tests:

Code

```yaml
steps:
  - ...

  - name: Install dependencies
    env:
      PYTHON_VERSION: ${{ matrix.python-version }}
    run: |
      python --version
      pip install -e .[tests]
      pip freeze
      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
```

Continuous Integration (CI)

Linting With the CI

Linting With the CI

The linting job should preferably be started before the tests.

Code

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

      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"

      - name: Check README syntax
        run: |
          pip install restructuredtext-lint
          restructuredtext-lint README.rst

      - uses: pre-commit/action@v3.0.1
        with:
          extra_args: --files $(git diff origin/main --name-only)
```

Continuous Integration (CI)

Building the Docs With the CI

Building the Docs With the CI

The docs job can be started last.

Code

```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
```

Continuous Integration (CI)

Changelog CI

Changelog CI

Code

```yaml
name: Changelog

on:
  pull_request:
    # should also be re-run when changing labels
    types: [opened, reopened, labeled, unlabeled, synchronize]

env:
  FRAGMENT_NAME: "docs/changes/${{ github.event.number }}.*.rst"

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

      - name: Check for news fragment
        if: ${{ ! contains( github.event.pull_request.labels.*.name, 'no-changelog-needed')}}
        uses: andstor/file-existence-action@v3
        with:
          files: ${{ env.FRAGMENT_NAME }}
          fail: true
```

Continuous Integration (CI)

GitLab CI

Let's take what we have written in GitHub Actions to GitLab CI:

1. Create a `.gitlab-ci.yml` file in the root of your repository
2. Basic setup:

Code

```yaml
image: condaforge/miniforge3:24.11.3-0

variables:
  MPLBACKEND: Agg
  PYTEST_ADDOPTS: --color=yes

stages:
  - lint
  - test
  - docs
```

3. Create a reusable template for all platforms:

Code

```
.test_template: &test_template
  stage: test
  before_script:
    - apt-get update && apt-get install -y curl bzip2

    - mamba env create -f environment.yml
    - source /opt/conda/etc/profile.d/conda.sh
    - conda activate [your_env_name]

    - pip install pytest-cov restructuredtext-lint pytest-xdist 'coverage!=6.3.0'
    - pip install -e .
    - pip freeze
    - pip list
  script:
    - pytest -vv --cov --cov-report=xml
  artifacts:
    paths:
      - coverage.xml
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage.xml
```

4. Add jobs for each platform:

Code

```
test:python-3.10:
  <<: *test_template
  variables:
    PYTHON_VERSION: "3.10"
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event" && $CI_MERGE_REQUEST_LABELS =~
    ↪  /documentation-only/
      when: never
    - when: always

test:python-3.11:
  <<: *test_template
  variables:
    PYTHON_VERSION: "3.11"
  after_script:
    - curl -Os https://cli.codecov.io/latest/linux/codecov
    - chmod +x codecov
    - ./codecov upload-process -t $CODECOV_TOKEN
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event" && $CI_MERGE_REQUEST_LABELS =~
    ↪  /documentation-only/
      when: never
    - when: always
```

Code

```yaml
lint:
  stage: lint
  image: python:3.12-slim
  before_script:
    - apt update -y
    - apt install -y git
    - git fetch --unshallow || true

    - pip install restructuredtext-lint pre-commit
    - pre-commit install
  script:
    - restructuredtext-lint README.rst
    - pre-commit run --files $(git diff origin/main --name-only) || true
  rules:
    - when: always
```

Test Docs Build | GitLab CI

Code

```yaml
docs:
  stage: test
  image: python:3.12-slim
  before_script:
    - apt update -y
    - apt install -y git build-essential pandoc graphviz ffmpeg make

    - pip install -U pip towncrier setuptools
    - pip install -e .[docs]

    - git describe --tags
  script:
    - make -C docs html
  rules:
    - when: always
```

Code

```yaml
changelog:
  stage: test
  image: ubuntu:latest
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
  variables:
    FRAGMENT_NAME: "docs/changes/${CI_MERGE_REQUEST_IID}.*.rst"
  before_script:
    - apt-get update && apt-get install -y git
  script:
    - git fetch --unshallow || true

    - |
      if echo "$CI_MERGE_REQUEST_LABELS" | grep -q "no-changelog-needed"; then
        echo "Skipping changelog check due to 'no-changelog-needed' label"
        exit 0
      fi

    - |
      if ls $FRAGMENT_NAME 1> /dev/null 2>&1; then
        echo "Changelog fragment found: $(ls $FRAGMENT_NAME)"
      else
        echo "Error: No changelog fragment found matching pattern: $FRAGMENT_NAME"
        echo "Please add a changelog fragment for this merge request."
        exit 1
      fi
  only:
    - merge_requests
```

Continuous Integration (CI)

Badges

Badges

- Badges are small images that display the status of your CI or code coverage
- Including a badge in your README is as simple as adding an image:

Code

```
============================
package |ci| |codecov| |pypi|
============================

.. |ci| image:: https://github.com/your_project/package/actions/workflows/ci.yml/badge.svg?branch=main
   :target: https://github.com/your_project/package/actions/workflows/ci.yml?branch=main
   :alt: Test Status

.. Badges from GitLab CI
.. |ci| image:: https://gitlab.com/your_project/package/badges/-/pipeline.svg
   :target: https://gitlab.com/your_project/package/-/pipelines/latest
   :alt: Test Status

.. |codecov| image:: https://codecov.io/github/your_project/package/badge.svg
   :target: https://codecov.io/github/your_project/package
   :alt: Code coverage

.. |pypi| image:: https://badge.fury.io/py/package.svg
   :target: https://pypi.org/project/package
   :alt: PyPI release
```

- Great sources for creating badges (e.g., PyPI version): badge.fury.io and shields.io

Summary

Summary | Docs

1. Write docs so users of your package are able to understand its API
2. If possible, provide tutorials and guides on how to get started
3. Ensure that all functions and classes have docstrings

Summary | CI

1. CIs are great tools to test and lint your code on the remote server
2. They should be used in tandem with tests on your **local** machine (see also `pre-commit`)
3. Depending on your requirements, some work is necessary to setup everything (e.g., job runners)

Backup

Sphinx Extensions: Notable Mentions

- `nbsphinx`: Built on `nbconvert` to include Jupyter Notebooks in the docs
- `sphinx-design`: Adds components for responsive web components
- `sphinx-gallery`: Creates example galleries from python scripts.

Jupyter Notebooks in Sphinx

Docs: nbsphinx

- Sometimes it is nicer to write a tutorial in a Jupyter notebook
- `nbsphinx` is a Sphinx extension that will include notebooks in your docs:
  ```
  $ mamba install nbsphinx
  $ pip install nbsphinx
  ```
- Add `nbsphinx` to your extensions list in `conf.py`

Code

```
extensions = [
    ...
    "nbsphinx"
]
```

- Create a new directory, e.g., `tutorials`, move the notebook (e.g., `plot.ipynb`) there, and add an `index.rst`:

Code

```
*********
Tutorials
*********

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

   plot .. name of the notebook
```