Hypermodern Python Cookiecutter

Claudio Jolowicz

Jun 03, 2022

CONTENTS

CHAPTER

ONE

QUICKSTART GUIDE

1.1 Requirements

Install Cookiecutter:

```
$ pipx install cookiecutter
```

Install Poetry by downloading and running install-poetry.py:

```
$ python install-poetry.py
```

Install Nox and nox-poetry:

```
$ pipx install nox
$ pipx inject nox nox-poetry
```

pipx is preferred, but you can also install with `pip install --user`.

It is recommended to set up Python 3.7, 3.8, 3.9, 3.10 using pyenv.

1.2 Creating a project

Generate a Python project:

```
$ cookiecutter gh:cjolowicz/cookiecutter-hypermodern-python \
  --checkout="2021.11.26"
```

Change to the root directory of your new project, and create a Git repository:

```
$ git init
$ git add .
$ git commit
```

1.3 Running

Run the command-line interface from the source tree:

```
$ poetry install
$ poetry run <project>
```

Run an interactive Python session:

```
$ poetry install
$ poetry run python
```

1.4 Testing

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

Install the pre-commit hooks:

```
$ nox -s pre-commit -- install
```

1.5 Continuous Integration

1.5.1 GitHub

1. Sign up at GitHub.
2. Create an empty repository for your project.
3. Follow the instructions to push an existing repository from the command line.

1.5.2 PyPI

1. Sign up at PyPI.
2. Go to the Account Settings on PyPI, generate an API token, and copy it.
3. Go to the repository settings on GitHub, and add a secret named `PYPI_TOKEN` with the token you just copied.

1.5.3 TestPyPI

1. Sign up at TestPyPI.
2. Go to the Account Settings on TestPyPI, generate an API token, and copy it.
3. Go to the repository settings on GitHub, and add a secret named `TEST_PYPI_TOKEN` with the token you just copied.

1.5.4 Codecov

1. Sign up at Codecov.
2. Install their GitHub app.

1.5.5 Read the Docs

1. Sign up at Read the Docs.
2. Import your GitHub repository, using the button *Import a Project*.
3. Install the GitHub webhook, using the button *Add integration* on the *Integrations* tab in the *Admin* section of your project on Read the Docs.

1.6 Releasing

Releases are triggered by a version bump on the default branch. It is recommended to do this in a separate pull request:

1. Switch to a branch.
2. Bump the version using poetry version.
3. Commit and push to GitHub.
4. Open a pull request.
5. Merge the pull request.

The Release workflow performs the following automated steps:

- Build and upload the package to PyPI.
- Apply a version tag to the repository.
- Publish a GitHub Release.

Release notes are populated with the titles and authors of merged pull requests. You can group the pull requests into separate sections by applying labels to them, like this:

Pull Request Label	Section in Release Notes
`breaking`	Breaking Changes
`enhancement`	Features
`removal`	Removals and Deprecations
`bug`	Fixes
`performance`	Performance
`testing`	Testing
`ci`	Continuous Integration
`documentation`	Documentation
`refactoring`	Refactoring
`style`	Style
`dependencies`	Dependencies

CHAPTER

TWO

USER GUIDE

This is the user guide for the Hypermodern Python Cookiecutter, a Python template based on the Hypermodern Python article series.

If you're in a hurry, check out the *quickstart guide* and the *tutorials*.

2.1 Introduction

2.1.1 About this project

The *Hypermodern Python Cookiecutter* is a general-purpose template for Python libraries and applications, released under the MIT license and hosted on GitHub.

The main objective of this project template is to enable current best practices through modern Python tooling. Our goals are to:

- focus on simplicity and minimalism,
- promote code quality through automation, and
- provide reliable and repeatable processes,

all the way from local testing to publishing releases.

Projects are created from the template using Cookiecutter, a project scaffolding tool built on top of the Jinja template engine.

The project template is centered around the following tools:

- Poetry for packaging and dependency management
- Nox for automation of checks and other development tasks
- GitHub Actions for continuous integration and delivery

2.1.2 Features

Here is a detailed list of features for this Python template:

- Packaging and dependency management with Poetry
- Test automation with Nox
- Linting with pre-commit and Flake8
- Continuous integration with GitHub Actions
- Documentation with Sphinx, MyST, and Read the Docs using the furo theme
- Automated uploads to PyPI and TestPyPI
- Automated release notes with Release Drafter
- Automated dependency updates with Dependabot
- Code formatting with Black and Prettier
- Import sorting with isort
- Testing with pytest
- Code coverage with Coverage.py
- Coverage reporting with Codecov
- Command-line interface with Click
- Static type-checking with mypy
- Runtime type-checking with Typeguard
- Automated Python syntax upgrades with pyupgrade
- Security audit with Bandit and Safety
- Check documentation examples with xdoctest
- Generate API documentation with autodoc and napoleon
- Generate command-line reference with sphinx-click
- Manage project labels with GitHub Labeler

The template supports Python 3.7, 3.8, 3.9, and 3.10.

2.1.3 Version policy

The *Hypermodern Python Cookiecutter* uses Calendar Versioning with a `YYYY.MM.DD` versioning scheme.

The current stable release is 2021.11.26.

2.2 Installation

2.2.1 System requirements

You need a recent Windows, Linux, Unix, or Mac system with git installed.

Note: When working with this template on Windows, configure your text editor or IDE to use only UNIX-style line endings (line feeds).

The project template contains a .gitattributes file which enables end-of-line normalization for your entire working tree. Additionally, the Prettier code formatter converts line endings to line feeds. Windows-style line endings (`CRLF`) should therefore never make it into your Git repository.

Nonetheless, configuring your editor for line feeds is recommended to avoid complaints from the pre-commit hook for Prettier.

2.2.2 Getting Python (Windows)

If you're on Windows, download the recommended installer for the latest stable release of Python from the official Python website. Before clicking **Install now**, enable the option to add Python to your `PATH` environment variable.

Verify your installation by checking the output of the following commands in a new terminal window:

```
python -VV
py -VV
```

Both of these commands should display the latest Python version, 3.10.

For local testing with multiple Python versions, repeat these steps for the latest bugfix releases of Python 3.7+, with the following changes:

- Do *not* enable the option to add Python to the `PATH` environment variable.
- `py -VV` and `python -VV` should still display the version of the latest stable release.
- `py -X.Y -VV` (e.g. `py -3.7 -VV`) should display the exact version you just installed.

Note that binary installers are not provided for security releases.

2.2.3 Getting Python (Mac, Linux, Unix)

If you're on a Mac, Linux, or Unix system, use pyenv for installing and managing Python versions. Please refer to the documentation of this project for detailed installation and usage instructions. (The following instructions assume that your system already has bash and curl installed.)

Install pyenv like this:

```
$ curl https://pyenv.run | bash
```

Add the following lines to your `~/.bashrc`:

```
export PATH="$HOME/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

Install the Python build dependencies for your platform, using one of the commands listed in the official instructions.

Install the latest point release of every supported Python version. This project template supports Python 3.7, 3.8, 3.9, and 3.10.

```
$ pyenv install 3.7.12
$ pyenv install 3.8.12
$ pyenv install 3.9.10
$ pyenv install 3.10.2
```

After creating your project (see *below*), you can make these Python versions accessible in the project directory, using the following command:

```
$ pyenv local 3.10.2 3.9.10 3.8.12 3.7.12
```

The first version listed is the one used when you type plain `python`. Every version can be used by invoking `python<major.minor>`. For example, use `python3.7` to invoke Python 3.7.

2.2.4 Requirements

Note: It is recommended to use pipx to install Python tools which are not specific to a single project. Please refer to the official documentation for detailed installation and usage instructions. If you decide to skip `pipx` installation, use pip install with the `--user` option instead.

You need four tools to use this template:

- Cookiecutter to create projects from the template,
- Poetry to manage packaging and dependencies
- Nox to automate checks and other tasks
- nox-poetry for using Poetry in Nox sessions

Install Cookiecutter using pipx:

```
$ pipx install cookiecutter
```

Install Poetry by downloading and running install-poetry.py:

```
$ python install-poetry.py
```

Install Nox and nox-poetry using pipx:

```
$ pipx install nox
$ pipx inject nox nox-poetry
```

Remember to upgrade these tools regularly:

```
$ pipx upgrade cookiecutter
$ pipx upgrade --include-injected nox
$ poetry self update
```

2.3 Project creation

2.3.1 Creating a project

Create a project from this template by pointing Cookiecutter to its GitHub repository. Use the `--checkout` option with the current stable release:

```
$ cookiecutter gh:cjolowicz/cookiecutter-hypermodern-python \
  --checkout="2021.11.26"
```

Cookiecutter downloads the template, and asks you a series of questions about project variables, for example, how you wish your project to be named. When you have answered these questions, your project is generated in the current directory, using a subdirectory with the same name as your project.

Here is a complete list of the project variables defined by this template:

Table 1: Project variables

Variable	Description	Example
`project_name`	Project name on PyPI and GitHub	`hypermodern-python`
`package_name`	Import name of the package	`hypermodern_python`
`friendly_name`	Friendly project name	`Hypermodern Python`
`author`	Primary author	Katherine Johnson
`email`	E-mail address of the author	katherine@example.com
`github_user`	GitHub username of the author	`katherine`
`version`	Initial project version	`0.0.0`
`copyright_year`	The project copyright year	`2022`
`license`	The project license	`MIT`
`development_status`	Development status of the project	`Development Status ::  3 - Alpha`

Note: The initial project version should be the latest release on PyPI, or `0.0.0` for an unreleased package. See *The Release workflow* for details.

Your choices are recorded in the file `.cookiecutter.json` in the generated project, together with the URL of this Cookiecutter template. Having this JSON file in the project makes it possible later on to update your project with changes from the Cookiecutter template, using tools such as cupper.

In the remainder of this guide, <project> and <package> are used to refer to the project and package names, respectively. By default, their only difference is that the project name uses hyphens (*kebab case*), whereas the package name uses underscores (*snake case*).

2.3.2 Uploading to GitHub

This project template is designed for use with GitHub. After generating the project, your next steps are to create a Git repository and upload it to GitHub.

Change to the root directory of your new project, initialize a Git repository, and create a commit for the initial project structure. In the commands below, replace <project> by the name of your project.

```
$ cd <project>
$ git init
```

(continues on next page)

(continued from previous page)

```
$ git add .
$ git commit
```

Use the following command to ensure your default branch is called `main`, which is the default branch name for GitHub repositories.

```
$ git branch --move --force main
```

Create an empty repository on GitHub, using the project name you chose when you generated the project.

Note: Do not include a `README.md`, `LICENSE`, or `.gitignore`. These files are provided by the project template.

Finally, upload your repository to GitHub. In the commands below, replace `<username>` by your GitHub username, and `<project>` by the name of your project.

```
$ git remote add origin git@github.com:<username>/<project>.git
$ git push --set-upstream origin main
```

Now may be a good time to set up Continuous Integration for your repository. Refer to the section *External services* for detailed instructions.

2.4 Project overview

2.4.1 Files and directories

This section provides an overview of all the files generated for your project.

Let's start with the directory layout:

Table 2: Directories

`src/<package>`	Python package
`tests`	Test suite
`docs`	Documentation
`.github/workflows`	GitHub Actions workflows

The Python package is located in the `src/<package>` directory. For more details on these files, refer to the section *The initial package*.

Table 3: Python package

`src/<project>/py.typed`	Marker file for PEP 561
`src/<project>/__init__.py`	Package initialization
`src/<project>/__main__.py`	Command-line interface

The test suite is located in the `tests` directory. For more details on these files, refer to the section *The test suite*.

Table 4: Test suite

`tests/__init__.py`	Test package initialization
`tests/test_main.py`	Test cases for `__main__`

The project documentation is written in Markdown. The documentation files in the top-level directory are rendered on GitHub:

Table 5: Documentation files (top-level)

`README.md`	Project description for GitHub and PyPI
`CONTRIBUTING.md`	Contributor Guide
`CODE_OF_CONDUCT.md`	Code of Conduct
`LICENSE`	License

The files in the `docs` directory are built using *Sphinx* and MyST. The Sphinx documentation is hosted on *Read the Docs*:

Table 6: Documentation files (Sphinx)

`index.md`	Main document
`contributing.md`	Contributor Guide (via include)
`codeofconduct.md`	Code of Conduct (via include)
`license.md`	License (via include)
`reference.md`	API reference
`usage.md`	Command-line reference

The `.github/workflows` directory contains the *GitHub Actions workflows*:

Table 7: GitHub Actions workflows

`release.yml`	*The Release workflow*
`tests.yml`	*The Tests workflow*
`labeler.yml`	*The Labeler workflow*

The project contains many configuration files for developer tools. Most of these are located in the top-level directory. The table below lists these files, and links each file to a section with more details.

Table 8: Configuration files

`.cookiecutter.json`	*Project variables*
`.darglint`	Configuration for *darglint*
`.github/dependabot.yml`	Configuration for *Dependabot*
`.flake8`	Configuration for *Flake8*
`.gitattributes`	Git attributes
`.gitignore`	Git ignore file
`.github/release-drafter.yml`	Configuration for *Release Drafter*
`.github/labels.yml`	Configuration for *GitHub Labeler*
`.pre-commit-config.yaml`	Configuration for *pre-commit*
`.readthedocs.yml`	Configuration for *Read the Docs*
`codecov.yml`	Configuration for *Codecov*
`docs/conf.py`	Configuration for *Sphinx*
`noxfile.py`	Configuration for *Nox*
`pyproject.toml`	Configuration for *Poetry*, *Coverage.py*, *isort*, and *mypy*

The `pyproject.toml` file is described in more detail *below*.

Dependencies are managed by Poetry and declared in the *pyproject.toml* file. The table below lists some additional files with pinned dependencies. Follow the links for more details on these.

Table 9: Dependency files

`poetry.lock`	*Poetry lock file*
`docs/requirements.txt`	Requirements file for *Read the Docs*
`.github/workflows/constraints.txt`	Constraints file for *GitHub Actions workflows*

2.4.2 The initial package

You can find the initial Python package in your generated project under the `src` directory:

```
src
└── <package>
    ├── __init__.py
    ├── __main__.py
    └── py.typed
```

`__init__.py` This file declares the directory as a Python package, and contains any package initialization code.

`__main__.py` The `__main__` module defines the entry point for the command-line interface. The command-line interface is implemented using the Click library, and supports `--help` and `--version` options. When the package is installed, a script named `<project>` is placed in the Python installation or virtual environment. This allows you to invoke the command-line interface using only the project name:

```
$ poetry run <project>  # during development
$ <project>             # after installation
```

The command-line interface can also be invoked by specifying a Python interpreter and the package name:

```
$ python -m <package> [<options>]
```

`py.typed` This is an empty marker file, which declares that your package supports typing and is distributed with its own type information (PEP 561). This allows people using your package to type-check their Python code against it.

2.4.3 The test suite

Tests are written using the pytest testing framework, the *de facto* standard for testing in Python.

The test suite is located in the `tests` directory:

```
tests
├── __init__.py
└── test_main.py
```

The test suite is declared as a package, and mirrors the source layout of the package under test. The file `test_main.py` contains tests for the `__main__` module.

Initially, the test suite contains a single test case, checking whether the program exits with a status code of zero. It also provides a test fixture using click.testing.CliRunner, a helper class for invoking the program from within tests.

For details on how to run the test suite, refer to the section *The tests session*.

2.4.4 Documentation

The project documentation is written in Markdown and processed by the Sphinx documentation engine using the MyST extension.

The top-level directory contains several stand-alone documentation files:

README.md This file is your main project page and displayed on GitHub and PyPI.

CONTRIBUTING.md The Contributor Guide explains how other people can contribute to your project.

CODE_OF_CONDUCT.md The Code of Conduct outlines the behavior expected from participants of your project. It is adapted from the Contributor Covenant, version 2.1.

LICENSE.md This file contains the text of your project's license.

Note: The files above are also rendered on GitHub and PyPI. Keep them in plain Markdown, without MyST syntax extensions.

The documentation files in the `docs` directory are built using Sphinx and MyST:

index.md This is the main documentation page. It includes the project description from `README.md`. This file also defines the navigation menu, with links to other documentation pages. The *Changelog* menu entry links to the GitHub Releases page of your project.

contributing.md This file includes the Contributor Guide from `CONTRIBUTING.md`.

codeofconduct.md This file includes the Code of Conduct from `CODE_OF_CONDUCT.md`.

license.md This file includes the license from `LICENSE.md`.

reference.md The API reference for your project. It is generated from docstrings and type annotations in the source code, using the autodoc and napoleon extensions.

usage.md The command-line reference for your project. It is generated by inspecting the click entry-point in your package, using the sphinx-click extension.

The `docs` directory contains two more files:

conf.py This Python file contains the Sphinx configuration.

requirements.txt The requirements file pins the build dependencies for the Sphinx documentation. This file is only used on Read the Docs.

The project documentation is built and hosted on *Read the Docs*, and uses the furo Sphinx theme.

You can also build the documentation locally using Nox, see *The docs session*.

2.5 Packaging

2.5.1 The pyproject.toml file

The configuration file for the Python package is located in the root directory of the project, and named `pyproject.toml`. It uses the TOML configuration file format, and contains two sections—*tables* in TOML parlance—, specified in PEP 517 and 518:

- The `build-system` table declares the requirements and the entry point used to build a distribution package for the project. This template uses Poetry as the build system.
- The `tool` table contains sub-tables where tools can store configuration under their PyPI name.

Table 10: Tool configurations in pyproject.toml

`tool.coverage`	Configuration for Coverage.py
`tool.isort`	Configuration for isort
`tool.mypy`	Configuration for mypy
`tool.poetry`	Configuration for Poetry

The `tool.poetry` table contains the metadata for your package, such as its name, version, and authors, as well as the list of dependencies for the package. Please refer to the Poetry documentation for a detailed description of each configuration key.

2.5.2 Version constraints

TL;DR

This project template omits upper bounds from all version constraints.

You are encouraged to manually remove upper bounds for dependencies you add to your project using Poetry:

1. Replace `^1.2.3` with `>=1.2.3` in `pyproject.toml`
2. Run `poetry lock --no-update` to update `poetry.lock`

Version constraints express which versions of dependencies are compatible with your project. In the case of core dependencies, they are also a part of distribution packages, and as such affect end-users of your package.

Note: Dependencies are Python packages used by your project, and they come in two types:

- *Core dependencies* are required by users running your code, and typically consist of third-party libraries imported by your package. When your package is distributed, the package metadata includes these dependencies, allowing tools like pip to automatically install them alongside your package.
- *Development dependencies* are only required by developers working on your code. Examples are applications used to run tests, check code for style and correctness, or to build documentation. These dependencies are not a part of distribution packages, because users do not require them to run your code.

For every dependency added to your project, Poetry writes a version constraint to `pyproject.toml`. Dependencies are kept in two TOML tables:

- `tool.poetry.dependencies`—for core dependencies
- `tool.poetry.dev-dependencies`—for development dependencies

By default, version constraints added by Poetry have both a lower and an upper bound:

- The lower bound requires users of your package to have at least the version that was current when you added the dependency.
- The upper bound allows users to upgrade to newer releases of dependencies, as long as the version number does not indicate a breaking change.

According to the Semantic Versioning standard, only major releases may contain breaking changes, once a project has reached version 1.0.0. A major release is one that increments the major version (the first component of the version identifier). An example for such a version constraint would be `^1.2.3`, which is a Poetry-specific shorthand equivalent to `>= 1.2.3, < 2`.

This project template omits upper bounds from all version constraints, in a conscious departure from Poetry's defaults. There are two separate reasons for removing version caps, one principled, the other pragmatic:

1. Version caps lead to problems in the Python ecosystem due to its flat dependency management.
2. Version caps lead to frequent merge conflicts between dependency updates.

The first point is treated in detail in the following articles:

- Should You Use Upper Bound Version Constraints? and Poetry Versions by Henry Schreiner
- Semantic Versioning Will Not Save You by Hynek Schlawack
- Version numbers: how to use them? by Bernát Gábor
- Why I don't like SemVer anymore by Brett Cannon

The second point is ultimately due to the fact that every updated version constraint changes a hashsum in the `poetry.lock` file. This means that PRs updating version constraints will *always* conflict with each other.

Note: The problem with merge conflicts is greatly exacerbated by a Dependabot issue: Dependabot updates version constraints in `pyproject.toml` even when the version constraint already covered the new version. This can be avoided using a configuration setting where only the lock file is ever updated, not the version constraints. Omitting version caps makes the lockfile-only strategy a viable alternative.

Poetry will still add `^1.2.3`-style version constraints whenever you add a dependency. You should edit the version constraint in `pyproject.toml`, replacing `^1.2.3` with `>=1.2.3` to remove the upper bound. Then update the lock file by invoking `poetry lock --no-update`.

2.5.3 The lock file

Poetry records the exact version of each direct and indirect dependency in its lock file, named `poetry.lock` and located in the root directory of the project. The lock file does not affect users of the package, because its contents are not included in distribution packages.

The lock file is useful for a number of reasons:

- It ensures that local checks run in the same environment as on the CI server, making the CI predictable and deterministic.
- When collaborating with other developers, it allows everybody to use the same development environment.
- When deploying an application, the lock file helps you keep production and development environments as similar as possible (dev-prod parity).

For these reasons, the lock file should be kept under source control.

2.5.4 Dependencies

This project template has a core dependency on Click, a library for creating command-line interfaces. The template also comes with various development dependencies. See the table below for an overview of the dependencies of generated projects:

Table 11: Dependencies

black	The uncompromising code formatter.
click	Composable command line interface toolkit
coverage	Code coverage measurement for Python
darglint	A utility for ensuring Google-style docstrings stay up to date with the source code.
flake8	the modular source code checker: pep8 pyflakes and co
flake8-bandit	Automated security testing with bandit and flake8.
flake8-bugbear	A plugin for flake8 finding likely bugs and design problems in your program.
flake8-docstrings	Extension for flake8 which uses pydocstyle to check docstrings
flake8-rst-docstrings	Python docstring reStructuredText (RST) validator
furo	A clean customisable Sphinx documentation theme.
isort	A Python utility / library to sort Python imports.
mypy	Optional static typing for Python
pep8-naming	Check PEP-8 naming conventions, plugin for flake8
pre-commit	A framework for managing and maintaining multi-language pre-commit hooks.
pre-commit-hooks	Some out-of-the-box hooks for pre-commit.
pygments	Pygments is a syntax highlighting package written in Python.
pytest	pytest: simple powerful testing with Python
pyupgrade	A tool to automatically upgrade syntax for newer versions.
safety	Checks installed dependencies for known vulnerabilities.
sphinx	Python documentation generator
sphinx-autobuild	Rebuild Sphinx documentation on changes, with live-reload in the browser.
sphinx-click	Sphinx extension that automatically documents click applications
typeguard	Run-time type checker for Python
xdoctest	A rewrite of the builtin doctest module

2.6 Using Poetry

Poetry manages packaging and dependencies for Python projects.

2.6.1 Managing dependencies

Use the command poetry show to see the full list of direct and indirect dependencies of your package:

```
$ poetry show
```

Use the command poetry add to add a dependency for your package:

```
$ poetry add foobar         # for core dependencies
$ poetry add --dev foobar   # for development dependencies
```

Important: It is recommended to remove the upper bound from the version constraint added by Poetry:

1. Edit `pyproject.toml` to replace `^1.2.3` with `>=1.2.3` in the dependency entry
2. Update `poetry.lock` using the command `poetry lock --no-update`

See *Version constraints* for more details.

Use the command poetry remove to remove a dependency from your package:

```
$ poetry remove foobar
```

Use the command poetry update to upgrade the dependency to a new release:

```
$ poetry update foobar
```

Note: Dependencies in the *Hypermodern Python Cookiecutter* are managed by *Dependabot*. When newer versions of dependencies become available, Dependabot updates the `poetry.lock` file and submits a pull request.

2.6.2 Installing the package for development

Poetry manages a virtual environment for your project, which contains your package, its core dependencies, and the development dependencies. All dependencies are kept at the versions specified by the lock file.

Note: A virtual environment gives your project an isolated runtime environment, consisting of a specific Python version and an independent set of installed Python packages. This way, the dependencies of your current project do not interfere with the system-wide Python installation, or other projects you're working on.

You can install your package and its dependencies into Poetry's virtual environment using the command poetry install.

```
$ poetry install
```

This command performs a so-called editable install of your package: Instead of building and installing a distribution package, it creates a special `.egg-link` file that links to your local source code. This means that code edits are directly visible in the environment without the need to reinstall your package.

Installing your package implicitly creates the virtual environment if it does not exist yet, using the currently active Python interpreter, or the first one found which satisfies the Python versions supported by your project.

2.6.3 Managing environments

You can create environments explicitly with the poetry env command, specifying the desired Python version. This allows you to create an environment for every Python version supported by your project, and easily switch between them:

```
$ poetry env use 3.7
$ poetry env use 3.8
$ poetry env use 3.9
$ poetry env use 3.10
```

Only one Poetry environment can be active at any time. Note that `3.10` comes last, to ensure that the current Python release is the active environment. Install your package with `poetry install` into each environment after creating it.

Use the command `poetry env list` to list the available environments:

```
$ poetry env list
```

Use the command `poetry env remove` to remove an environment:

```
$ poetry env remove <version>
```

Use the command `poetry env info` to show information about the active environment:

```
$ poetry env info
```

2.6.4 Running commands

You can run an interactive Python session inside the active environment using the command poetry run:

```
$ poetry run python
```

The same command allows you to invoke the command-line interface of your project:

```
$ poetry run <project>
```

You can also run developer tools, such as pytest:

```
$ poetry run pytest
```

While it is handy to have developer tools available in the Poetry environment, it is usually recommended to run these using Nox, as described in the section *Using Nox*.

2.6.5 Building and distributing the package

Note: With the *Hypermodern Python Cookiecutter*, building and distributing your package is taken care of by GitHub Actions. For more information, see the section *The Release workflow*.

This section gives a short overview of how you can build and distribute your package from the command line, using the following Poetry commands:

```
$ poetry build
$ poetry publish
```

Building the package is done with the python build command, which generates *distribution packages* in the `dist` directory of your project. These are compressed archives that an end-user can download and install on their system. They come in two flavours: source (or *sdist*) archives, and binary packages in the wheel format.

Publishing the package is done with the python publish command, which uploads the distribution packages to your account on PyPI, the official Python package registry.

2.6.6 Installing the package

Once your package is on PyPI, others can install it with pip, pipx, or Poetry:

```
$ pip install <project>
$ pipx install <project>
$ poetry add <project>
```

While pip is the workhorse of the Python packaging ecosystem, you should use higher-level tools to install your package:

- If the package is an application, install it with pipx.
- If the package is a library, install it with poetry add in other projects.

The primary benefit of these installation methods is that your package is installed into an isolated environment, without polluting the system environment, or the environments of other applications. This way, applications can use specific versions of their direct and indirect dependencies, without getting in each other's way.

If the other project is not managed by Poetry, use whatever package manager the other project uses. You can always install your project into a virtual environment with plain pip.

2.7 Using Nox

Nox automates testing in multiple Python environments. Like its older sibling tox, Nox makes it easy to run any kind of job in an isolated environment, with only those dependencies installed that the job needs.

Nox sessions are defined in a Python file named `noxfile.py` and located in the project directory. They consist of a virtual environment and a set of commands to run in that environment.

While Poetry environments allow you to interact with your package during development, Nox environments are used to run developer tools in a reliable and repeatable way across Python versions.

Most sessions are run with every supported Python version. Other sessions are only run with the current stable Python version, for example the session used to build the documentation.

2.7.1 Running sessions

If you invoke Nox by itself, it will run the full test suite:

```
$ nox
```

This includes tests, linters, type checks, and more. For the full list, please refer to the table *below*.

The list of sessions run by default can be configured by editing `nox.options.sessions` in `noxfile.py`. Currently the list only excludes the *docs session* (which spawns an HTTP server) and the *coverage session* (which is triggered by the *tests session*).

You can also run a specific Nox session, using the `--session` option. For example, build the documentation like this:

```
$ nox --session=docs
```

Print a list of the available Nox sessions using the `--list-sessions` option:

```
$ nox --list-sessions
```

Nox creates virtual environments from scratch on each invocation. You can speed things up by passing the –reuse-existing-virtualenvs option, or the equivalent short option `-r`. For example, the following may be more practical during development (this will only run tests and type checks, on the current Python release):

```
$ nox -p 3.10 -rs tests mypy
```

Many sessions accept additional options after `--` separator. For example, the following command runs a specific test module:

```
$ nox --session=tests -- tests/test_main.py
```

2.7.2 Overview of Nox sessions

The following table gives an overview of the available Nox sessions:

Table 12: Nox sessions

Session	Description	Python	Default
coverage	Report coverage with Coverage.py	`3.10`	(✓)
docs	Build and serve Sphinx documentation	`3.10`	
docs-build	Build Sphinx documentation	`3.10`	✓
mypy	Type-check with mypy	`3.7` ... `3.10`	✓
pre-commit	Lint with pre-commit	`3.10`	✓
safety	Scan dependencies with Safety	`3.10`	✓
tests	Run tests with pytest	`3.7` ... `3.10`	✓
typeguard	Type-check with Typeguard	`3.10`	✓
xdoctest	Run examples with xdoctest	`3.7` ... `3.10`	✓

2.7.3 The docs session

Build the documentation using the Nox session `docs`:

```
$ nox --session=docs
```

The docs session runs the command `sphinx-autobuild` to generate the HTML documentation from the Sphinx directory. This tool has several advantages over `sphinx-build` when you are editing the documentation files:

- It rebuilds the documentation whenever a change is detected.
- It spins up a web server with live reloading.
- It opens the location of the web server in your browser.

Use the `--` separator to pass additional options. For example, to treat warnings as errors and run in nit-picky mode:

```
$ nox --session=docs -- -W -n docs docs/_build
```

This Nox session always runs with the current major release of Python.

2.7.4 The docs-build session

The `docs-build` session runs the command `sphinx-build` to generate the HTML documentation from the Sphinx directory.

This session is meant to be run as a part of automated checks. Use the interactive `docs` session instead while you're editing the documentation.

This Nox session always runs with the current major release of Python.

2.7.5 The mypy session

mypy is the pioneer and *de facto* reference implementation of static type checking in Python. Learn more about it in the section *Type-checking with mypy*.

Run mypy using Nox:

```
$ nox --session=mypy
```

You can also run the type checker with a specific Python version. For example, the following command runs mypy using the current stable release of Python:

```
$ nox --session=mypy --python=3.10
```

Use the separator `--` to pass additional options and arguments to `mypy`. For example, the following command type-checks only the `__main__` module:

```
$ nox --session=mypy -- src/<package>/__main__.py
```

2.7.6 The pre-commit session

pre-commit is a multi-language linter framework and a Git hook manager. Learn more about it in the section *Linting with pre-commit*.

Run pre-commit from Nox using the `pre-commit` session:

```
$ nox --session=pre-commit
```

This session always runs with the current stable release of Python.

Use the separator `--` to pass additional options to `pre-commit`. For example, the following command installs the pre-commit hooks, so they run automatically on every commit you make:

```
$ nox --session=pre-commit -- install
```

2.7.7 The safety session

Safety checks the dependencies of your project for known security vulnerabilities, using a curated database of insecure Python packages. The *Hypermodern Python Cookiecutter* uses the poetry export command to convert Poetry's lock file to a requirements file, for consumption by Safety.

Run Safety using the `safety` session:

```
$ nox --session=safety
```

This session always runs with the current stable release of Python.

2.7.8 The tests session

Tests are written using the pytest testing framework. Learn more about it in the section *The test suite*.

Run the test suite using the Nox session `tests`:

```
$ nox --session=tests
```

The tests session runs the test suite against the installed code. More specifically, the session builds a wheel from your project and installs it into the Nox environment, with dependencies pinned as specified by Poetry's lock file.

You can also run the test suite with a specific Python version. For example, the following command runs the test suite using the current stable release of Python:

```
$ nox --session=tests --python=3.10
```

Use the separator `--` to pass additional options to `pytest`. For example, the following command runs only the test case `test_main_succeeds`:

```
$ nox --session=tests -- -k test_main_succeeds
```

The tests session also installs pygments, a Python syntax highlighter. It is used by pytest to highlight code in tracebacks, improving the readability of test failures.

2.7.9 The coverage session

Note: *Test coverage* is a measure of the degree to which the source code of your program is executed while running its test suite.

The coverage session prints a detailed coverage report to the terminal, combining the coverage data collected during the *tests session*. If the total coverage is below 100%, the coverage session fails. Code coverage is measured using Coverage.py.

The coverage session is triggered by the tests session, and runs after all other sessions have completed. This allows it to combine the coverage data for different Python versions.

You can also run the session manually:

```
$ nox --session=coverage
```

Use the `--` separator to pass arguments to the `coverage` command. For example, here's how you would generate an HTML report in the `htmlcov` directory:

```
$ nox -rs coverage -- html
```

Coverage.py is configured in the `pyproject.toml` file, using the `tool.coverage` table. The configuration informs the tool about your package name and source tree layout. It also enables branch analysis and the display of line numbers for missing coverage, and specifies the target coverage percentage. Coverage is measured for the package as well as the test suite itself.

During continuous integration, coverage data is uploaded to the Codecov reporting service. For details, see the sections about *Codecov* and *The Tests workflow*.

2.7.10 The typeguard session

Typeguard is a runtime type checker and pytest plugin. It can type-check function calls during test runs via an import hook.

Typeguard checks that arguments passed to functions match the type annotations of the function parameters, and that the return value provided by the function matches the return type annotation. In the case of generator functions, Typeguard checks the yields, sends and the return value against the `Generator` annotation.

Run Typeguard using Nox:

```
$ nox --session=typeguard
```

The typeguard session runs the test suite with runtime type-checking enabled. It is similar to the *tests session*, with the difference that your package is instrumented by Typeguard.

This session always runs with the current stable release of Python.

Use the separator `--` to pass additional options and arguments to pytest. For example, the following command runs only tests for the `__main__` module:

```
$ nox --session=typeguard -- tests/test_main.py
```

Note: Typeguard generates a warning about missing type annotations for a Click object. This is due to the fact that `__main__.main` is wrapped by a decorator, and its type annotations only apply to the inner function, not the resulting object as seen by the test suite.

2.7.11 The xdoctest session

The xdoctest tool runs examples in your docstrings and compares the actual output to the expected output as per the docstring. This serves multiple purposes:

- The example is checked for correctness.
- You ensure that the documentation is up-to-date.
- Your codebase gets additional test coverage for free.

Run the tool using the Nox session `xdoctest`:

```
$ nox --session=xdoctest
```

You can also run the test suite with a specific Python version. For example, the following command runs the examples using the current stable release of Python:

```
$ nox --session=xdoctest --python=3.10
```

By default, the Nox session uses the `all` subcommand to run all examples. You can also list examples using the `list` subcommand, or run specific examples:

```
$ nox --session=xdoctest -- list
```

2.8 Linting with pre-commit

pre-commit is a multi-language linter framework and a Git hook manager. It allows you to integrate linters and formatters into your Git workflow, even when written in a language other than Python.

pre-commit is configured using the file `.pre-commit-config.yaml` in the project directory. Please refer to the official documentation for details about the configuration file.

2.8.1 Running pre-commit from Nox

pre-commit runs in a Nox session every time you invoke `nox`:

```
$ nox
```

Run the pre-commit session explicitly like this:

```
$ nox --session=pre-commit
```

The session is described in more detail in the section *The pre-commit session*.

2.8.2 Running pre-commit from git

When installed as a Git hook, pre-commit runs automatically every time you invoke `git commit`. The commit is aborted if any check fails. When invoked in this mode, pre-commit only runs on files staged for the commit.

Install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

2.8.3 Managing hooks with pre-commit

Hooks in languages other than Python, such as `prettier`, run in isolated environments managed by pre-commit. To upgrade these hooks, use the autoupdate command:

```
$ nox --session=pre-commit -- autoupdate
```

2.8.4 Python-language hooks

Note: This section provides some background information about how this project template integrates pre-commit with Poetry and Nox. You can safely skip this section.

Python-language hooks in the *Hypermodern Python Cookiecutter* are not managed by pre-commit. Instead, they are tracked as development dependencies in Poetry, and installed into the Nox session alongside pre-commit itself. As development dependencies, they are also present in the Poetry environment.

This approach has some advantages:

- All project dependencies are managed by Poetry.
- Hooks receive automatic upgrades from Dependabot.

- Nox can serve as a single entry point for all checks.
- Additional hook dependencies can be upgraded by a dependency manager. An example for this are Flake8 extensions. By contrast, `pre-commit autoupdate` does not include additional dependencies.
- Dependencies of dependencies (*subdependencies*) can be locked automatically, making checks more repeatable and deterministic.
- Linters and formatters are available in the Poetry environment, which is useful for editor integration.

There are also some drawbacks to this technique:

- This is not the officially supported way to integrate pre-commit hooks.
- The hook scripts installed by pre-commit do not activate the virtual environment in which pre-commit and the hooks are installed. To work around this limitation, the Nox session patches hook scripts on installation.
- Adding a hook is more work, including updating `pyproject.toml` and `noxfile.py`, and adding the hook definition to `pre-commit-config.yaml`.

You can always opt out of this integration method, by removing the `repo: local` section from the configuration file, and adding the official pre-commit hooks instead. Don't forget to remove the hooks from Poetry's dependencies and from the Nox session.

Note: Python-language hooks in the *Hypermodern Python Cookiecutter* are defined as system hooks. System hooks don't have their environments managed by pre-commit; instead, pre-commit assumes that hook dependencies have already been installed and are available in its environment. The Nox session for pre-commit takes care of installing the Python hooks alongside pre-commit.

Furthermore, the *Hypermodern Python Cookiecutter* defines Python-language hooks as repository-local hooks. As such, hook definitions are not supplied by the hook repositories, but by the project itself. This makes it possible to override the hook language to `system`, as explained above.

2.8.5 Adding an official pre-commit hook

Adding the official pre-commit hook for a linter is straightforward. Often you can simply copy a configuration snippet from the repository's `README`. Otherwise, note the hook identifier from the `pre-commit-hooks.yaml` file, and the git tag for the latest version. Add the following section to your `pre-commit-config.yaml`, under `repos`:

```
- repo: <hook repository>
  rev: <version tag>
  hooks:
    - id: <hook identifier>
```

While this technique also works for Python-language hooks, it is recommended to integrate Python hooks with Nox and Poetry, as shown in the next section.

2.8.6 Adding a Python-language hook

Adding a Python-language hook to your project takes three steps:

- Add the hook as a Poetry development dependency.
- Install the hook in the Nox session for pre-commit.
- Add the hook to `pre-commit-config.yaml`.

For example, consider a linter named `awesome-linter`.

First, use Poetry to add the linter to your development dependencies:

```
$ poetry add --dev awesome-linter
```

Next, update `noxfile.py` to add the linter to the pre-commit session:

```
@nox.session(name="pre-commit", ...)
def precommit(session: Session) -> None:
    ...
    session.install(
        "awesome-linter",  # Install awesome-linter
        "black",
        "darglint",
        ...
    )
```

Finally, add the hook to `pre-commit-config.yaml` as follows:

- Locate the `pre-commit-hooks.yaml` file in the `awesome-linter` repository.
- Copy the entry for the hook (not just the hook identifier).
- Change `language:` from `python` to `system`.
- Add the hook definition to the `repo: local` section.

Depending on the linter, the hook definition might look somewhat like the following:

```
repos:
  - repo: local
    hooks:
      # ...
      - id: awesome-linter
        name: Awesome Linter
        entry: awesome-linter
        language: system # was: python
        types: [python]
```

2.8.7 Running checks on modified files

pre-commit runs checks on the *staged* contents of files. Any local modifications are stashed for the duration of the checks. This is motivated by pre-commit's primary use case, validating changes staged for a commit.

Requiring changes to be staged allows for a nice property: Many pre-commit hooks support fixing offending lines automatically, for example `black`, `prettier`, and `isort`. When this happens, your original changes are in the staging area, while the fixes are in the work tree. You can accept the fixes by staging them with `git add` before committing again.

If you want to run linters or formatters on modified files, and you do not want to stage the modifications just yet, you can also invoke the tools via Poetry instead. For example, use `poetry run flake8 <file>` to lint a modified file with Flake8.

2.8.8 Overview of pre-commit hooks

The *Hypermodern Python Cookiecutter* comes with a pre-commit configuration consisting of the following hooks:

Table 13: pre-commit hooks

black	Run the Black code formatter
flake8	Run the Flake8 linter
isort	Rewrite source code to sort Python imports
prettier	Run the Prettier code formatter
pyupgrade	Upgrade syntax to newer versions of Python
check-added-large-files	Prevent giant files from being committed
check-toml	Validate TOML files
check-yaml	Validate YAML files
end-of-file-fixer	Ensure files are terminated by a single newline
trailing-whitespace	Ensure lines do not contain trailing whitespace

2.8.9 The Black hook

Black is the uncompromising Python code formatter. One of its greatest features is its lack of configurability. Blackened code looks the same regardless of the project you're reading.

2.8.10 The Prettier hook

Prettier is an opinionated code formatter for many languages, including YAML, Markdown, and JavaScript. Like Black, it has few options, and the *Hypermodern Python Cookiecutter* uses none of them.

2.8.11 The Flake8 hook

Flake8 is an extensible linter framework for Python. For more details, see the section *Linting with Flake8*.

2.8.12 The isort hook

isort reorders imports in your Python code. Imports are separated into three sections, as recommended by PEP 8: standard library, third party, first party. There are two additional sections, one at the top for future imports, the other at the bottom for relative imports. Within each section, `from` imports follow normal imports. Imports are then sorted alphabetically.

The *Hypermodern Python Cookiecutter* activates the Black profile for compatibility with the Black code formatter. Furthermore, the force_single_line setting is enabled. This splits imports onto separate lines to avoid merge conflicts. Finally, two blank lines are enforced after imports for consistency, via the lines_after_imports setting.

2.8.13 The pyupgrade hook

pyupgrade upgrades your source code to newer versions of the Python language and standard library. The tool analyzes the abstract syntax tree of the modules in your project, replacing deprecated or legacy usages with modern idioms.

The minimum supported Python version is declared in the relevant section of `.pre-commit-config.yaml`. You should change this setting whenever you drop support for an old version of Python.

2.8.14 Hooks from pre-commit-hooks

The pre-commit configuration also includes several smaller hooks from the pre-commit-hooks repository.

2.9 Linting with Flake8

Flake8 is an extensible linter framework for Python, and a command-line utility to run the linters on your source code. The *Hypermodern Python Cookiecutter* integrates Flake8 via a pre-commit hook, see the section *The Flake8 hook*.

The configuration file for Flake8 and its extensions is named `.flake8` and located in the project directory. For details about the configuration file, see the official reference.

The sections below describe the linters in more detail. Each section also notes any configuration settings applied by the *Hypermodern Python Cookiecutter*.

2.9.1 Overview of available plugins

Flake8 comes with a rich ecosystem of plugins. The following table lists the Flake8 plugins used by the *Hypermodern Python Cookiecutter*, and links to their lists of error codes.

Table 14: Flake8 plugins

pyflakes	Find invalid Python code	F
pycodestyle	Enforce style conventions from PEP 8	E,W
pep8-naming	Enforce naming conventions from PEP 8	N
pydocstyle / flake8-docstrings	Enforce docstring conventions from PEP 257	D
flake8-rst-docstrings	Find invalid reStructuredText in docstrings	RST
flake8-bugbear	Detect bugs and design problems	B
mccabe	Limit the code complexity	C
darglint	Detect inaccurate docstrings	DAR
Bandit / flake8-bandit	Detect common security issues	S

2.9.2 pyflakes

pyflakes parses Python source files and finds invalid code. Warnings reported by this tool include syntax errors, undefined names, unused imports or variables, and more. It is included with Flake8 by default.

Error codes are prefixed by `F` for "flake".

2.9.3 pycodestyle

pycodestyle checks your code against the style recommendations of PEP 8, the official Python style guide. The tool detects whitespace and indentation issues, deprecated features, bare excepts, and much more. It is included with Flake8 by default.

Error codes are prefixed by `W` for warnings and `E` for errors.

The *Hypermodern Python Cookiecutter* disables the following errors and warnings for compatibility with Black and flake8-bugbear:

- `E203` (whitespace before `:`)
- `E501` (line too long)
- `W503` (line break before binary operator)

2.9.4 pep8-naming

pep8-naming enforces the naming conventions from PEP 8. Examples are the use of camel case for the names of classes, the use of lowercase for the names of functions, arguments and variables, or the convention to name the first argument of methods `self`.

Error codes are prefixed by `N` for "naming".

2.9.5 pydocstyle and flake8-docstrings

pydocstyle checks that docstrings comply with the recommendations of PEP 257 and a configurable style convention. It is integrated with Flake8 via the flake8-docstrings extension. Warnings range from missing docstrings to issues with whitespace, quoting, and docstring content.

Error codes are prefixed by `D` for "docstring".

The *Hypermodern Python Cookiecutter* selects the recommendations of the Google styleguide. Here is an example of a function documented in Google style:

```
def add(first: int, second: int) -> int:
    """Add two integers.

    Args:
        first: The first argument.
        second: The second argument.

    Returns:
        The sum of the arguments.
    """
```

2.9.6 flake8-rst-docstrings

flake8-rst-docstrings validates docstring markup as reStructuredText. Docstrings must be valid reStructuredText because they are used by Sphinx to generate the API reference.

Error codes are prefixed by `RST` for "reStructuredText", and group issues into numerical blocks, by their severity and origin.

2.9.7 flake8-bugbear

flake8-bugbear detects bugs and design problems. The warnings are more opinionated than those of pyflakes or pycodestyle. For example, the plugin detects Python 2 constructs which have been removed in Python 3, and likely bugs such as function arguments defaulting to empty lists or dictionaries.

Error codes are prefixed by `B` for "bugbear".

The *Hypermodern Python Cookiecutter* also enables Bugbear's `B9` warnings, which are disabled by default. In particular, `B950` checks the maximum line length like pycodestyle's `E501`, but with a tolerance margin of 10%. This soft limit is set to 80 characters, which is the value used by the Black code formatter.

2.9.8 mccabe

mccabe checks the code complexity of your Python package against a configured limit. The tool is included with Flake8.

Error codes are prefixed by `C` for "complexity".

The *Hypermodern Python Cookiecutter* limits code complexity to a value of 10.

2.9.9 darglint

darglint checks that docstring descriptions match function definitions. The tool has its own configuration file, named `.darglint`.

Error codes are prefixed by `DAR` for "darglint".

The *Hypermodern Python Cookiecutter* allows one-line docstrings without function signatures. Multi-line docstrings must specify the function signatures completely and correctly, using Google docstring style.

2.9.10 Bandit

Bandit is a tool designed to find common security issues in Python code, and integrated via the flake8-bandit extension.

Error codes are prefixed by `S` for "security". (The prefix `B` for "bandit" is used when Bandit is run as a stand-alone tool.)

The *Hypermodern Python Cookiecutter* disables `S101` (use of assert) for the test suite, as pytest uses assertions to verify expectations in tests.

2.10 Type-checking with mypy

Note: Type annotations, first introduced in Python 3.5, are a way to annotate functions and variables with types. With appropriate tooling, they can make your programs easier to understand, debug, and maintain.

Type-checking refers to the practice of verifying the type correctness of a program, using type annotations and type inference. There are two kinds of type checkers:

- *Static type checkers* verify the type correctness of your program without executing it, using static analysis.
- *Runtime type checkers* find type errors by instrumenting your code to type-check arguments and return values in function calls. This is particularly useful during the execution of unit tests.

There is also an increasing number of libraries that leverage type annotations at runtime. For example, you can use type annotations to generate serialization schemas or command-line parsers.

mypy is the pioneer and *de facto* reference implementation of static type checking in Python. Invoke mypy via Nox, as explained in the section *The mypy session*.

mypy is configured in the `pyproject.toml` file, using the `tool.mypy` table. For details about supported configuration options, see the official reference.

The *Hypermodern Python Cookiecutter* enables several configuration options which are off by default. The following options are enabled for strictness and enhanced output:

- `strict`
- `warn_unreachable`
- `pretty`
- `show_column_numbers`
- `show_error_codes`
- `show_error_context`

2.11 External services

Your GitHub repository can be integrated with several external services for continuous integration and delivery. This section describes these external services, what they do, and how to set them up for your repository.

2.11.1 PyPI

PyPI is the official Python Package Index. Uploading your package to PyPI allows others to download and install it to their system.

Follow these steps to set up PyPI for your repository:

1. Sign up at PyPI.
2. Go to the Account Settings on PyPI, generate an API token, and copy it.
3. Go to the repository settings on GitHub, and add a secret named `PYPI_TOKEN` with the token you just copied.

PyPI is integrated with your repository via the *Release workflow*.

2.11.2 TestPyPI

TestPyPI is a test instance of the Python package registry. It allows you to check your release before uploading it to the real index.

Follow these steps to set up TestPyPI for your repository:

1. Sign up at TestPyPI.
2. Go to the Account Settings on TestPyPI, generate an API token, and copy it.
3. Go to the repository settings on GitHub, and add a secret named `TEST_PYPI_TOKEN` with the token you just copied.

TestPyPI is integrated with your repository via the *Release workflow*.

2.11.3 Codecov

Codecov is a reporting service for code coverage.

Follow these steps to set up Codecov for your repository:

1. Sign up at Codecov.
2. Install their GitHub app.

The configuration is included in the repository, in the file codecov.yml.

Codecov integrates with your repository via its GitHub app. The *Tests workflow* uploads the coverage data.

2.11.4 Dependabot

Dependabot creates pull requests with automated dependency updates.

Please refer to the official documentation for more details.

The configuration is included in the repository, in the file .github/dependabot.yml.

It manages the following dependencies:

Type of dependency	Managed files	See also
Python	`poetry.lock`	*Managing dependencies*
Python	`docs/requirements.txt`	*Read the Docs*
Python	`.github/workflows/constraints.txt`	*Constraints file*
GitHub Action	`.github/workflows/*.yml`	*GitHub Actions workflows*

2.11.5 Read the Docs

Read the Docs automates the building, versioning, and hosting of documentation.

Follow these steps to set up Read the Docs for your repository:

1. Sign up at Read the Docs.
2. Import your GitHub repository, using the button *Import a Project*.
3. Install the GitHub webhook, using the button *Add integration* on the *Integrations* tab in the *Admin* section of your project on Read the Docs.

Read the Docs automatically starts building your documentation, and will continue to do so when you push to the default branch or make a release. Your documentation now has a public URL like this:

https://<project>.readthedocs.io/

The configuration for Read the Docs is included in the repository, in the file .readthedocs.yml. The *Hypermodern Python Cookiecutter* configures Read the Docs to build and install the package with Poetry, using a so-called PEP 517-build.

Build dependencies for the documentation are installed using a requirements file located at `docs/requirements.txt`. Read the Docs currently does not support installing development dependencies using Poetry's lock file. For the sake of brevity and maintainability, only direct dependencies are included.

Note: The requirements file is managed by *Dependabot*. When newer versions of the build dependencies become available, Dependabot updates the requirements file and submits a pull request. When adding or removing Sphinx extensions using Poetry, don't forget to update the requirements file as well.

2.12 GitHub Actions workflows

The *Hypermodern Python Cookiecutter* uses GitHub Actions to implement continuous integration and delivery. With GitHub Actions, you define so-called workflows using YAML files located in the `.github/workflows` directory.

A *workflow* is an automated process consisting of one or many jobs, each of which executes a series of steps. Workflows are triggered by events, for example when a commit is pushed or when a release is published. You can learn more about the workflow language and its supported keywords in the official reference.

Note: Real-time logs for workflow runs are available from the *Actions* tab in your GitHub repository.

2.12.1 Overview of workflows

The *Hypermodern Python Cookiecutter* defines the following workflows:

Table 15: GitHub Actions workflows

Workflow	File	Description	Trigger
Tests	`tests.yml`	Run the test suite with Nox	Push, PR
Release	`release.yml`	Upload the package to PyPI	Push (default branch)
Labeler	`labeler.yml`	Manage GitHub project labels	Push (default branch)

2.12.2 Overview of GitHub Actions

Workflows use the following GitHub Actions:

Table 16: GitHub Actions

actions/cache	Cache dependencies and build outputs
actions/checkout	Check out the Git repository
actions/download-artifact	Download artifacts from workflows
actions/setup-python	Set up workflows with a specific Python version
actions/upload-artifact	Upload artifacts from workflows
codecov/codecov-action	Upload coverage to Codecov
crazy-max/ghaction-github-labeler	Manage labels on GitHub as code
pypa/gh-action-pypi-publish	Upload packages to PyPI and TestPyPI
release-drafter/release-drafter	Draft and publish GitHub Releases
salsify/action-detect-and-tag-new-version	Detect and tag new versions in a repository

Note: GitHub Actions used by the workflows are managed by *Dependabot*. When newer versions of GitHub Actions become available, Dependabot updates the workflows that use them and submits a pull request.

2.12.3 Constraints file

GitHub Actions workflows install the following tools:

- pip
- virtualenv
- Poetry
- Nox

These dependencies are pinned using a constraints file located in `.github/workflow/constraints.txt`.

Note: The constraints file is managed by *Dependabot*. When newer versions of the tools become available, Dependabot updates the constraints file and submits a pull request.

2.12.4 The Tests workflow

The Tests workflow runs checks using Nox. It is triggered on every push to the repository, and when a pull request is opened or receives new commits.

Each Nox session runs in a separate job, using the current release of Python and the latest Ubuntu runner. Selected Nox sessions also run on Windows and macOS, and with older Python versions, as shown in the table below:

Table 17: Jobs in the Tests workflow

Nox session	Platform	Python versions
pre-commit	Ubuntu	3.10
safety	Ubuntu	3.10
mypy	Ubuntu	3.10, 3.9, 3.8, 3.7
tests	Ubuntu	3.10, 3.9, 3.8, 3.7
tests	Windows	3.10
tests	macOS	3.10
coverage	Ubuntu	3.10
docs-build	Ubuntu	3.10

The workflow uploads the generated documentation as a workflow artifact. Building the documentation only serves the purpose of catching issues in pull requests. Builds on Read the Docs happen independently.

The workflow also uploads coverage data to Codecov after running tests. It generates a coverage report in Cobertura XML format, using the *coverage session*. The report is uploaded using the official Codecov GitHub Action.

The Tests workflow uses the following GitHub Actions:

- actions/checkout for checking out the Git repository
- actions/setup-python for setting up the Python interpreter
- actions/download-artifact to download the coverage data of each tests session
- actions/cache for caching pre-commit environments
- actions/upload-artifact to upload the generated documentation and the coverage data of each tests session
- codecov/codecov-action for uploading to Codecov

The Tests workflow is defined in `.github/workflows/tests.yml`.

2.12.5 The Release workflow

The Release workflow publishes your package on PyPI, the Python Package Index. The workflow also creates a version tag in the GitHub repository, and publishes a GitHub Release using Release Drafter. The workflow is triggered on every push to the default branch.

Release steps only run if the package version was bumped. If the package version did not change, the package is instead uploaded to TestPyPI as a prerelease, and only a draft GitHub Release is created. TestPyPI is a test instance of the Python Package Index.

The Release workflow uses API tokens to access PyPI and TestPyPI. You can generate these tokens from your account settings on these services. The tokens need to be stored as secrets in the repository settings on GitHub:

Table 18: Secrets

`PYPI_TOKEN`	PyPI API token
`TEST_PYPI_TOKEN`	TestPyPI API token

The Release workflow uses the following GitHub Actions:

- actions/checkout for checking out the Git repository
- actions/setup-python for setting up the Python interpreter
- salsify/action-detect-and-tag-new-version for tagging on version bumps
- pypa/gh-action-pypi-publish for uploading the package to PyPI or TestPyPI
- release-drafter/release-drafter for publishing the GitHub Release

Release notes are populated with the titles and authors of merged pull requests. You can group the pull requests into separate sections by applying labels to them, like this:

Pull Request Label	Section in Release Notes
`breaking`	Breaking Changes
`enhancement`	Features
`removal`	Removals and Deprecations
`bug`	Fixes
`performance`	Performance
`testing`	Testing
`ci`	Continuous Integration
`documentation`	Documentation
`refactoring`	Refactoring
`style`	Style
`dependencies`	Dependencies

The workflow is defined in `.github/workflows/release.yml`. The Release Drafter configuration is located in `.github/release-drafter.yml`.

2.12.6 The Labeler workflow

The Labeler workflow manages the labels used in GitHub issues and pull requests based on a description file `.github/labels.yaml`. In this file each label is described with a `name`, a `description` and a `color`. The workflow is triggered on every push to the default branch.

The workflow creates or updates project labels if they are missing or different compared to the `labels.yml` file content.

The workflow does not delete labels already configured in the GitHub UI and not in the `labels.yml` file. You can change this behavior and add ignore patterns in the settings of the workflow (see GitHub Labeler documentation).

The Labeler workflow uses the following GitHub Actions:

- actions/checkout for checking out the Git repository
- crazy-max/ghaction-github-labeler for updating the GitHub project labels

The workflow is defined in `.github/workflows/labeler.yml`. The GitHub Labeler configuration is located in `.github/labels.yml`.

2.13 Tutorials

First, make sure you have all the *requirements* installed.

2.13.1 How to test your project

Run the test suite using *Nox*:

```
$ nox -r
```

2.13.2 How to run your code

First, install the project and its dependencies to the Poetry environment:

```
$ poetry install
```

Run an interactive session in the environment:

```
$ poetry run python
```

Invoke the command-line interface of your package:

```
$ poetry run <project>
```

2.13.3 How to make code changes

1. Run the tests, *as explained above*. All tests should pass.
2. Add a failing test *under the tests directory*. Run the tests again to verify that your test fails.
3. Make your changes to the package, *under the src directory*. Run the tests to verify that all tests pass again.

2.13.4 How to push code changes

Create a branch for your changes:

```
$ git switch --create my-topic-branch main
```

Create a series of small, single-purpose commits:

```
$ git add <files>
$ git commit
```

Push your branch to GitHub:

```
$ git push --set-upstream origin my-topic-branch
```

The push triggers the following automated steps:

- *The test suite runs against your branch*.

2.13.5 How to open a pull request

Open a pull request for your branch on GitHub:

1. Select your branch from the *Branch* menu.
2. Click **New pull request**.
3. Enter the title for the pull request.
4. Enter a description for the pull request.
5. Apply a *label identifying the type of change*
6. Click **Create pull request**.

Release notes are pre-filled with the titles of merged pull requests.

2.13.6 How to accept a pull request

If all checks are marked as passed, merge the pull request using the squash-merge strategy (recommended):

1. Click **Squash and Merge**. (Select this option from the dropdown menu of the merge button, if it is not shown.)
2. Click **Confirm squash and merge**.
3. Click **Delete branch**.

This triggers the following automated steps:

- *The test suite runs against the main branch*.
- *The draft GitHub Release is updated*.
- *A pre-release of the package is uploaded to TestPyPI*.
- Read the Docs rebuilds the *latest* version of the documentation.

In your local repository, update the main branch:

```
$ git switch main
$ git pull origin main
```

Optionally, remove the merged topic branch from the local repository as well:

```
$ git remote prune origin
$ git branch --delete --force my-topic-branch
```

The original commits remain accessible from the pull request (*Commits* tab).

2.13.7 How to make a release

Releases are triggered by a version bump on the default branch. It is recommended to do this in a separate pull request:

1. Switch to a branch.
2. Bump the version using poetry version.
3. Commit and push to GitHub.
4. Open a pull request.
5. Merge the pull request.

The individual steps for bumping the version are:

```
$ git switch --create release main
$ poetry version <version>
$ git commit --message="<project> <version>" pyproject.toml
$ git push origin release
```

If you're not sure which version number to choose, read about Semantic Versioning. Versioning rules for Python packages are laid down in PEP 440.

Before merging the pull request for the release, go through the following checklist:

- The pull request passes all checks.
- The development release on TestPyPI looks good.
- All pull requests for the release have been merged.

Merging the pull request triggers the *Release workflow*. This workflow performs the following automated steps:

- Publish the package on PyPI.
- Publish a GitHub Release.
- Apply a Git tag to the repository.

Read the Docs automatically builds a new stable version of the documentation.

2.14 The Hypermodern Python blog

The project setup is described in detail in the Hypermodern Python article series:

- Chapter 1: Setup
- Chapter 2: Testing
- Chapter 3: Linting
- Chapter 4: Typing
- Chapter 5: Documentation
- Chapter 6: CI/CD

You can also read the articles on this blog.

CHAPTER

THREE

CONTRIBUTOR GUIDE

Thank you for your interest in improving the Hypermodern Python Cookiecutter. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

3.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

3.2 How to request a feature

Request features on the Issue Tracker.

3.3 How to set up your development environment

You need Python 3.7+ and the following tools:

- Cookiecutter
- Poetry
- Nox
- nox-poetry

Fork the repository on GitHub, and clone the fork to your local machine. You can now generate a project from your development version:

```
$ cookiecutter path/to/cookiecutter-hypermodern-python
```

You may also want to push your generated project to GitHub, and set up continuous integration.

3.4 How to test the project

Please refer to the User Guide for instructions on how to run the test suite locally.

3.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

3.6 How to accept changes

You need to be a project maintainer to accept changes.

Before accepting a pull request, go through the following checklist:

- The PR must pass all checks.
- The PR must have a descriptive title.
- The PR should be labelled with the kind of change (see below).

Release notes are pre-filled with titles and authors of merged pull requests. Labels group the pull requests into sections. The following list shows the available sections, with associated labels in parentheses:

- Breaking Changes (`breaking`)

- Features (`enhancement`)
- Removals and Deprecations (`removal`)
- Fixes (`bug`)
- Performance (`performance`)
- Testing (`testing`)
- Continuous Integration (`ci`)
- Documentation (`documentation`)
- Refactoring (`refactoring`)
- Style (`style`)
- Dependencies (`dependencies`)

To merge the pull request, follow these steps:

1. Click **Squash and Merge**. (Select this option from the dropdown menu of the merge button, if it is not shown.)
2. Click **Confirm squash and merge**.
3. Click **Delete branch**.

3.7 How to make a release

You need to be a project maintainer to make a release.

Before making a release, go through the following checklist:

- All pull requests for the release have been merged.
- The default branch passes all checks.

Releases are made by publishing a GitHub Release. A draft release is being maintained based on merged pull requests. To publish the release, follow these steps:

1. Click **Edit** next to the draft release.
2. Enter a tag with the new version.
3. Enter the release title, also the new version.
4. Edit the release description, if required.
5. Click **Publish Release**.

Version numbers adhere to Calendar Versioning, of the form `YYYY.MM.DD`.

After publishing the release, the following automated steps are triggered:

- The Git tag is applied to the repository.
- Read the Docs builds a new stable version of the documentation.

CHAPTER

FOUR

CONTRIBUTOR COVENANT CODE OF CONDUCT

4.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

4.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

4.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

4.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at mail@claudiojolowicz.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

4.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

4.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

4.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

4.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

4.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

CHAPTER

FIVE

LICENSE

```
MIT License

Copyright © 2020 Claudio Jolowicz

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

Cookiecutter template for a Python package based on the Hypermodern Python article series.

CHAPTER

SIX

USAGE

```
$ cookiecutter gh:cjolowicz/cookiecutter-hypermodern-python \
  --checkout="2021.11.26"
```

CHAPTER

SEVEN

FEATURES

- Packaging and dependency management with Poetry
- Test automation with Nox
- Linting with pre-commit and Flake8
- Continuous integration with GitHub Actions
- Documentation with Sphinx, MyST, and Read the Docs using the furo theme
- Automated uploads to PyPI and TestPyPI
- Automated release notes with Release Drafter
- Automated dependency updates with Dependabot
- Code formatting with Black and Prettier
- Import sorting with isort
- Testing with pytest
- Code coverage with Coverage.py
- Coverage reporting with Codecov
- Command-line interface with Click
- Static type-checking with mypy
- Runtime type-checking with Typeguard
- Automated Python syntax upgrades with pyupgrade
- Security audit with Bandit and Safety
- Check documentation examples with xdoctest
- Generate API documentation with autodoc and napoleon
- Generate command-line reference with sphinx-click
- Manage project labels with GitHub Labeler

The template supports Python 3.7, 3.8, 3.9, and 3.10.

CHAPTER

EIGHT

FAQ

8.1 What is this project about?

The mission of this project is to enable current best practices through modern Python tooling.

8.2 What makes this project different from other Python templates?

This is a general-purpose template for Python libraries and applications.

Our goals are:

- Focus on simplicity and minimalism
- Promote code quality through automation
- Provide reliable and repeatable processes

The project template is centered around the following tools:

- Poetry for packaging and dependency management
- Nox for automation of checks and other development tasks
- GitHub Actions for continuous integration and delivery

8.3 Why is this Python template called “hypermodern”?

Hypermodernism is a school of chess that dates back to more than a century ago. If this setup ever goes out of fashion, I can pretend it was my secret plan from the start. All images on the associated blog show past visions of the future.