Hypermodern Python Cookiecutter

Claudio Jolowicz

Apr 22, 2020

CONTENTS

CHAPTER

ONE

QUICKSTART GUIDE

1.1 Requirements

Install Cookiecutter:

```
$ pipx install cookiecutter
```

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

```
$ python get-poetry.py
```

Install Nox:

```
$ pipx install nox
```

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

It is recommended to set up Python 3.6, 3.7, and 3.8 using pyenv.

1.2 Creating a project

Generate a Python project:

```
$ cookiecutter gh:cjolowicz/cookiecutter-hypermodern-python \
  --checkout="2020.4.15"
```

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

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

1.3 Local testing

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

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

1.4 Continuous Integration

1.4.1 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.

1.4.2 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.

1.4.3 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.

1.4.4 Codecov

1. Sign up at Codecov, and install their GitHub app.
2. Add your repository to Codecov.

1.4.5 Read the Docs

1. Sign up at Read the Docs.
2. Import your GitHub repository, using the button *Import a Project*.

1.5 Releasing

1. Bump the version using poetry version. Push to GitHub.
2. Publish a GitHub Release.
3. GitHub Action triggers the PyPI upload.

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

Use labels to group the pull requests into sections:

Label	Section
`breaking`	Breaking Changes
`enhancement`	Features
`removal`	Removals and Deprecations
`bug`	Fixes
`performance`	Performance
`testing`	Testing
`ci`	Continuous Integration
`documentation`	Documentation
`refactoring`	Refactoring
`style`	Style
`build`	Build System and Dependencies

GitHub creates the `bug`, `enhancement`, and `documentation` labels for you. Create the remaining labels on the Issues tab of your GitHub repository.

1.6 Caveats

When upgrading Sphinx or its extensions using Poetry, also update the requirements located in `docs/requirements.txt` for Read the Docs.

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 practises through modern Python tooling. Our goals are to:

- keep a 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
- Continuous integration with GitHub Actions
- Documentation with Sphinx and Read the Docs
- Automated uploads to PyPI and TestPyPI
- Automated release notes with Release Drafter
- Code formatting with Black and Prettier
- Testing with pytest
- Code coverage with Coverage.py
- Coverage reporting with Codecov
- Command-line interface with Click
- Linting with Flake8 and various *awesome plugins*
- Static type-checking with mypy and pytype
- Runtime type-checking with Typeguard
- Security audit with Bandit and Safety
- Git hook management with pre-commit
- Checked documentation examples with xdoctest

- API documentation with autodoc, napoleon, and sphinx-autodoc-typehints

The template supports Python 3.6, 3.7, and 3.8.

2.1.3 Release cadence

The *Hypermodern Python Cookiecutter* has a bimonthly release cadence. Releases happen on the 15th of every other month, starting in January. We use Calendar Versioning with a `YYYY.MM.DD` versioning scheme. Initial releases may occur more frequently.

The current stable release is 2020.4.15.

2.2 Installation

2.2.1 System requirements

You need a recent Linux, Unix, or Mac system with bash, curl, and git.

On Windows 10, enable the Windows Subsystem for Linux (WSL) and install the Ubuntu 18.04 LTS distribution. Open Ubuntu from the Start Menu, and install additional packages using the following commands:

```
$ sudo apt update
$ sudo apt install -y build-essential curl git libbz2-dev \
  libffi-dev liblzma-dev libncurses5-dev libncursesw5-dev \
  libreadline-dev libsqlite3-dev libssl-dev llvm make \
  python-openssl tk-dev wget xz-utils zlib1g-dev
```

The project template should also work natively on Windows. Pull requests to document Windows specifics are welcome!

2.2.2 Getting Python

It is recommended to use pyenv for installing and managing Python versions. Please refer to the documentation of this project for detailed installation and usage instructions.

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 3.6, 3.7, and 3.8.

```
$ pyenv install 3.6.10
$ pyenv install 3.7.7
$ pyenv install 3.8.2
```

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

```
$ pyenv local 3.8.2 3.7.7 3.6.10
```

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.

2.2.3 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 only need three 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

As an optional requirement, pre-commit is recommended for additional checks and to manage Git hooks.

Install Cookiecutter using pipx:

```
$ pipx install cookiecutter
```

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

```
$ python get-poetry.py
```

Install Nox using pipx:

```
$ pipx install nox
```

Install pre-commit using pipx:

```
$ pipx install pre-commit
```

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="2020.4.15"
```

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:

Project 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	Jane Doe
`email`	E-mail address of the author	jane.doe@example.com
`github_user`	GitHub username of the author	`janedoe`
`version`	Initial project version	`0.1.0`

In the remainder of this guide, `<project>` and `<package>` are used to refer to the project and package names, respectively.

2.3.2 The initial package

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

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

The `__init__.py` file declares the directory as a Python package. It also defines a `__version__` attribute, containing the version of your package. The version is determined using the installed package metadata, by means of the standard importlib.metadata library.

The `console.py` module defines the `console.main` entry point for the command-line interface. The command-line interface is implemented using Click, and supports `--help` and `--version` options. When the package is installed, a script named `<project>` is placed in the `bin` directory of the Python installation or virtual environment, allowing you to invoke the command-line interface like any other console application.

The `__main__.py` module allows you to invoke the command-line interface by specifying a Python interpreter and the package name:

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

2.3.3 Uploading to GitHub

This project template is designed for use with GitHub, so 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:

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

Create an empty repository on GitHub, using the project name you chose when you generated the project. 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 `<repository>` by the name of your GitHub repository.

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

2.4 Packaging

2.4.1 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. Poetry stores its configuration in the `tool.poetry` table.

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.

2.4.2 Building and distributing the package

Note: With the *Hypermodern Python Cookiecutter*, building and distributing your package is taken care of by GitHub Actions when you publish a GitHub Release.

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. This command generates *distribution packages* in the `dist` directory of your project. These are compressed archives which 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. This command uploads the distribution packages to your account on PyPI, the official Python package registry.

2.4.3 Installing the package

With your package 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 normally 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.

2.5 Dependencies

2.5.1 Types of dependencies

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. These dependencies are also declared in distribution packages such as wheels, 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.

This project template has two core dependencies:

- Click, a library for creating command-line interfaces
- importlib_metadata, a backport of importlib.metadata

The project template also comes with a large number of development dependencies. See *Features* for an overview.

2.5.2 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
```

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
```

To upgrade to a new major release, you normally need to update the version constraint for the dependency, in the `pyproject.toml` file.

2.5.3 Version constraints

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.

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 require users to have at least the version of a dependency that was current when you added it to the project. Users can also upgrade to newer releases of dependencies, as long as the version number does not indicate a breaking change. (After 1.0.0, Semantic Versioning limits breaking changes to major releases.)

2.5.4 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.

2.6 The Poetry environment

Poetry manages a virtual environment for your project, containing your package together with its core dependencies, as well as the development dependencies. All dependencies are kept at the versions specified by the lock file.

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.

2.6.1 Installing the package for development

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.

2.6.2 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 3.6
$ poetry env use 3.7
$ poetry env use 3.8
```

Only one Poetry environment can be active at any time. Note that `3.8` 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
```

2.6.3 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 *next* section.

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 unit tests, linters, and type checkers, but excludes sessions like those for building documentation or for reformatting code. The list of sessions run by default can be configured by editing `nox.options.sessions` in `noxfile.py`.

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 (a sensible default). You can speed things up by passing the –reuse-existing-virtualenvs option (or the equivalent short option `-r`):

```
$ nox --reuse-existing-virtualenvs
```

2.7.2 Available sessions

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

Session	Description	Python	Default
black	Format code with Black	`3.8`	
coverage	Generate a coverage report	`3.8`	
docs	Build Sphinx documentation	`3.8`	
lint	Lint with Flake8	`3.6` ... `3.8`	✓
mypy	Type-check with mypy	`3.6` ... `3.8`	✓
pytype	Type-check with pytype	`3.6` ... `3.7`	✓
safety	Scan dependencies with Safety	`3.8`	✓
tests	Run tests with pytest	`3.6` ... `3.8`	✓
typeguard	Type-check with Typeguard	`3.6` ... `3.8`	
xdoctest	Run examples with xdoctest	`3.6` ... `3.8`	

2.7.3 Using Poetry inside Nox sessions

Nox sessions can invoke Poetry like any other command, using the function nox.sessions.Session.run. Integrating Nox and Poetry in a sane way requires additional work. For this purpose, `noxfile.py` contains some glue code in the form of the `install` and `install_package` functions, and the `Poetry` helper class.

`noxfile.install(session, *args)`: Install dependencies into a Nox session using Poetry.

The `noxfile.install` function installs development dependencies into a Nox session, using the versions specified in Poetry's lock file. This is done by exporting the lock file in `requirements.txt` format, and passing it as a constraints file to pip. The function arguments are the same as those for nox.sessions.Session.install: The first argument is the `Session` object, and the remaining arguments are command-line arguments for pip install, typically just the package or packages to be installed.

noxfile.install_package(session): Install the package into a Nox session using Poetry.

The `noxfile.install_package` function installs your package into a Nox session, including the core dependencies as specified in Poetry's lock file. This is done by building a wheel from the package, and installing it using pip. Dependencies are installed in the same way as in the `noxfile.install` function, i.e. using a constraints file. Its only argument is the `Session` object from Nox.

The functions are implemented using a `Poetry` helper class, encapsulating invocations of the Poetry command-line interface. The helper class has the following methods:

noxfile.Poetry.build(self, *args) Build the package.

noxfile.Poetry.export(self, *args) Export the lock file to requirements format.

noxfile.Poetry.version(self) Return the package version.

noxfile.Poetry.__init__(self, session) Instances need a session object for running commands.

2.8 Testing

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

2.8.1 The test suite

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

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

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

2.8.2 The tests session

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 in 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-3.8
```

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
```

2.8.3 Test coverage

Test coverage is a measure of the degree to which the source code of your program is executed while running its test suite. This project template requires full test coverage.

Code coverage is measured using Coverage.py. When the test suite completes, a detailed coverage report is printed to the terminal. If the total coverage is below 100%, the test session fails.

Coverage.py is configured using the `pyproject.toml` configuration file, in 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.

2.8.4 The coverage session

Note: This session is intended for use inside Continuous Integration. For a coverage report, simply run the *tests* session.

Run the coverage session like this:

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

The coverage session exports the coverage data to cobertura XML format, the format expected by Codecov.

This session always runs with the current major release of Python. It does not accept additional options.

2.9 Linting with Flake8

This project template comes with an extensive suite of linters, using the Flake8 linter framework. Linters analyze source code to flag programming errors, bugs, stylistic errors, and suspicious constructs.

By default, the linter suite checks Python files in the following locations:

- `src`
- `tests`
- `noxfile.py`
- `docs/conf.py`

The configuration file for Flake8 and its extensions is named `.flake8` and located in the project directory.

2.9.1 The lint session

Run the linter suite using the `lint` session:

```
$ nox --session=lint
```

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

```
$ nox --session=lint-3.8
```

Use the separator `--` to pass additional options to `flake8`. For example, the following command only lints the `console` module:

```
$ nox --session=lint -- src/<project>/console.py
```

2.9.2 Available linters

Flake8 glues together several tools, and comes with a rich ecosystem of extensions. The following table lists the linters used by the *Hypermodern Python Cookiecutter*, and links to their lists of error codes.

Tool	Description	Code
pyflakes	Find invalid Python code	F
pycodestyle	Enforce style conventions from PEP 8	E,W
pep8-naming	Enforce naming conventions from PEP 8	N
flake8-import-order	Enforce import conventions from PEP 8	I
flake8-docstrings	Enforce docstring conventions from PEP 257, via pydocstyle	D
flake8-rst-docstrings	Find invalid reStructuredText in docstrings	RST
flake8-black	Enforce the Black code style	BLK
flake8-bugbear	Detect bugs and design problems	B
mccabe	Limit the code complexity	C
flake8-annotations	Enforce type coverage	ANN
darglint	Detect inaccurate docstrings	DAR
flake8-bandit	Detect common security issues, via Bandit	S

2.9.3 The linters

This section describes the linters in more detail. Each section also notes any configuration settings applied by the *Hypermodern Python Cookiecutter*.

pyflakes

The pyflakes tool parses Python source files and finds invalid code. Error codes are prefixed by `F` for "flake". Warnings reported by this tool include syntax errors, undefined names, unused imports or variables, and more. The tool is included with Flake8 by default.

pycodestyle

The pycodestyle tool checks your code against many recommendations from PEP 8, the official Python style guide. Error codes are prefixed by `W` for warnings and `E` for errors. The tool detects whitespace and indentation issues, deprecated features, bare excepts, and much more. The tool is included with Flake8 by default.

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)

pep8-naming

The pep8-naming tool enforces the naming conventions from PEP 8. Error codes are prefixed by `N` for "naming". 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`.

flake8-import-order

The flake8-import-order plugin checks that import order adheres to PEP 8 and a configurable style convention. Error codes are prefixed by `I` for "import".

The *Hypermodern Python Cookiecutter* selects the recommendations of the Google styleguide. Imports need to be arranged in three sorted groups, like this:

```
# standard library
import time

# third-party packages
import click

# local packages
import <package>
```

The configuration also ensures that the package name is recognized as local.

pydocstyle and flake8-docstrings

The pydocstyle tool is used to check that docstrings comply with the recommendations of PEP 257 and a configurable style convention. It is integrated via the flake8-docstrings extension. Error codes are prefixed by `D` for "docstring". Warnings range from missing docstrings to issues with whitespace, quoting, and docstring content.

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.
    """
```

flake8-rst-docstrings

The flake8-rst-docstrings plugin validates docstring markup as reStructuredText (reST). Docstrings must be valid reST—which includes most plain text—because they are used to generate API documentation. Error codes are prefixed by `RST` for "reStructuredText", and group issues into numerical blocks, by their severity and origin.

flake8-black

The flake8-black plugin checks adherence to the Black code style. Error codes are prefixed by `BLK` for "black". It generates a warning if it detects that Black would reformat a source file. You can fix these issues automatically, as described below in the section *Code formatting with Black*.

flake8-bugbear

flake8-bugbear detects bugs and design problems. Error codes are prefixed by `B` for "bugbear". 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.

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.

mccabe

The mccabe tool checks the code complexity of your Python package against a configured limit. Error codes are prefixed by `C` for "complexity". It is included with Flake8.

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

flake8-annotations

flake8-annotations detects the absence of type annotations for functions, helping you keep track of unannotated code. Error codes are prefixed by `ANN` for "annotation".

The *Hypermodern Python Cookiecutter* disables the warning `ANN101` (missing type annotation for `self` in method), because annotating `self` is normally not required.

darglint

The darglint tool checks that docstring descriptions match function definitions. Error codes are prefixed by `DAR` for "darglint". The tool has its own configuration file, named `.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.

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 Code formatting with Black

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.

The *Hypermodern Python Cookiecutter* adheres to Black code style.

2.10.1 The black session

Run the code formatter using the `black` session:

```
$ nox --session=black
```

This session always runs with the current version of Python.

Use the separator `--` to pass additional options to `black`. For example, the following command formats a specific file:

```
$ nox --session=black -- noxfile.py
```

By default, the code formatter runs on Python files in the following locations:

- `src`
- `tests`
- `noxfile.py`
- `docs/conf.py`

2.11 Scanning dependencies with Safety

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.

2.11.1 The safety session

Run Safety using the `safety` session:

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

This session always runs with the current version of Python.

2.12 Linting with pre-commit

pre-commit is a multi-language linter framework and a Git hook manager. It allows you to integrate the best industry standard linters into your Git workflow, even when written in a language other than Python. Linters run in isolated environments managed by pre-commit.

When installed as a *pre-commit* Git hook, pre-commit runs automatically every time you invoke `git commit`. The commit is aborted if any check fails. This workflow allows you to review the changes before attempting the commit again. Many linters support fixing offending lines automatically.

2.12.1 Installation and configuration

Install pre-commit via pipx:

```
$ pipx install pre-commit
```

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.

2.12.2 Available hooks

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

Hook	Description
black	Run the Black code formatter
flake8	Run the Flake8 linter
prettier	Run the Prettier code formatter
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

Black and Flake8 are managed as development dependencies by Poetry. Therefore, their hooks are run in the Poetry environment, rather than in pre-commit environments. These checks run somewhat faster than the corresponding Nox sessions, for several reasons:

- They only run on files staged for a commit, by default.
- They only run on the current version of Python.
- They assume that the tools are already installed.

2.12.3 Command-line usage

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

```
$ pre-commit install
```

The default behaviour of pre-commit is to run on the staged contents of files, which is useful when it is triggered from a *pre-commit* Git hook:

```
$ pre-commit run
```

You can run pre-commit on all files instead using the following command:

```
$ pre-commit run --all-files
```

You can also run a specific pre-commit hook, such as the code formatter Prettier:

```
$ pre-commit run --all-files prettier
```

2.13 Type-checking

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. 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.

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.

The *Hypermodern Python Cookiecutter* uses both static type checkers and a runtime type checker:

- mypy is the pioneer and *de facto* reference implementation of static type checking in Python.
- pytype is a static type checker developed at Google, with a focus on type inference and stub generation.
- Typeguard is a runtime type checker and pytest plugin. It can type-check function calls during test runs via an import hook.

2.13.1 The mypy session

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-3.8
```

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

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

Configure mypy using the mypy.ini configuration file.

The *Hypermodern Python Cookiecutter* disables import errors for some packages for which type definitions are not yet available, using the `ignore_missing_imports` option.

2.13.2 The pytype session

Run pytype using Nox:

```
$ nox --session=pytype
```

You can also run the type checker with a specific Python version. For example, the following command runs pytype using Python 3.7:

```
$ nox --session=pytype-3.7
```

pytype does not yet support Python 3.8.

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

```
$ nox --session=pytype -- --disable=import-error src/<package>/console.py
```

The command-line option `--disable=import-error` avoids errors for third-party packages without typing information. This option is passed by default if the session is run without additional arguments.

2.13.3 The typeguard session

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.

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` or `AsyncGenerator` annotation.

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

```
$ nox --session=typeguard-3.8
```

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

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

Typeguard generates a warning about missing type annotations for a Click object. This is due to the fact that `console.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.

2.14 Documentation

2.14.1 Markdown files

The project repository contains several documentation files written in Markdown or plain text:

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

2.14.2 Sphinx documentation

The project documentation itself lives under `docs`. It is written in reStructuredText, processed by Sphinx, and accessible on Read the Docs. It consists of the following files:

File	Contents
`conf.py`	Sphinx configuration file
`index.rst`	Master document
`license.rst`	License (included from `LICENSE`)
`reference.rst`	API documentation
`requirements.txt`	Build dependencies for Read the Docs

The Contributor Guide and Code of Conduct are included from the Markdown files via the recommonmark extension. The documentation menu also has a *Changelog* entry, which links to the GitHub Releases page.

The API documentation is generated from docstrings and type annotations, using the autodoc, napoleon, and sphinx-autodoc-typehints extensions.

The `requirements.txt` is necessary because Read the Docs currently does not support installing development dependencies using Poetry's lock file. You need to update this file manually, whenever you upgrade Sphinx or its extensions. For the sake of brevity and maintainability, only direct dependencies are listed.

2.14.3 The docs session

Build the documentation using the Nox session `docs`:

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

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

In interactive mode—such as when invoking Nox from a terminal—sphinx-autobuild is used instead. This tool has several advantages 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 to either tool. 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.

2.14.4 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-3.8
```

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
```

2.15 Continuous integration using GitHub Actions

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.

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

2.15.1 Secrets

Some workflows use tokens to access external services. The following table lists the required tokens, which need to be stored as secrets in the repository settings on GitHub:

Name	Description
`PYPI_TOKEN`	PyPI API token
`TEST_PYPI_TOKEN`	TestPyPI API token

You can generate these API tokens from your account settings on PyPI and TestPyPI.

2.15.2 GitHub Apps

Install the Codecov GitHub app, and add your repository to Codecov. The sign up process will guide you through these steps.

2.15.3 Available workflows

The *Hypermodern Python Cookiecutter* defines the following workflows:

Workflow	File	Description	Trigger
Tests	`tests.yml`	Run the test suite with Nox	Push
Coverage	`coverage.yml`	Upload coverage data to Codecov	Push
pre-commit	`pre-commit.yml`	Run linters with pre-commit	Push
Release Drafter	`release-drafter.yml`	Update the draft GitHub Release	Push (master)
Release	`release.yml`	Upload the package to PyPI	GitHub Release
TestPyPI	`test-pypi.yml`	Upload the package to TestPyPI	Push (master)

The Tests workflow

The Tests workflow executes the test suite using Nox.

The workflow is triggered on every push to the GitHub repository. It consists of a job for each supported Python version, running on the latest supported Ubuntu image.

The workflow uses the following GitHub Actions:

- actions/checkout for checking out the Git repository
- actions/setup-python for setting up the Python interpreter

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

The Coverage workflow

The Coverage workflow uploads coverage data to Codecov.

The workflow is triggered on every push to the GitHub repository. It executes the *tests session* to collect coverage data, and the *coverage session* to produce a coverage report in XML format. This coverage report is then uploaded to Codecov.

The workflow uses the following GitHub Actions:

- actions/checkout for checking out the Git repository
- actions/setup-python for setting up the Python interpreter
- codecov/codecov-action for uploading to Codecov

The workflow runs on the current Python version and the latest supported Ubuntu image.

It is defined in `.github/workflows/coverage.yml`.

The pre-commit workflow

The pre-commit workflow runs *pre-commit* on all files in the repository.

The workflow is triggered on every push to the GitHub repository.

The workflow uses the following GitHub Actions:

- actions/checkout for checking out the Git repository
- actions/setup-python for setting up the Python interpreter
- actions/cache for caching pre-commit environments

The workflow runs on the current Python version and the latest supported Ubuntu image.

It is defined in `.github/workflows/pre-commit.yml`.

The Release Drafter workflow

The Release Drafter workflow maintains a draft for the next GitHub Release.

The workflow is triggered on every push to the master branch. It includes details from every pull request merged into master since the last release. The workflow uses the Release Drafter GitHub Action.

The *Hypermodern Python Cookiecutter* groups pull requests by type, using GitHub labels. The following table shows the section headings and corresponding labels:

Label	Section
`breaking`	Breaking Changes
`enhancement`	Features
`removal`	Removals and Deprecations
`bug`	Fixes
`performance`	Performance
`testing`	Testing
`ci`	Continuous Integration
`documentation`	Documentation
`refactoring`	Refactoring
`style`	Style
`build`	Build System and Dependencies

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

The Release workflow

The Release workflow publishes your package on PyPI, the Python Package Index.

The workflow is triggered when a GitHub Release is published. It checks that the test suite passes, builds the package using Poetry, and uploads it using the pypa/gh-action-pypi-publish action. This workflow uses the `PYPI_TOKEN` secret.

The workflow is defined in `.github/workflows/release.yml`.

The TestPyPI workflow

The TestPyPI workflow publishes your package on TestPyPI, a test instance of the Python Package Index.

The workflow is triggered on every push to the master branch. It bumps the version number to a developmental pre-release, builds the package using Poetry, and uploads it using the pypa/gh-action-pypi-publish action. This workflow uses the `TEST_PYPI_TOKEN` secret.

The workflow is defined in `.github/workflows/test-pypi.yml`.

2.16 Read the Docs

Read the Docs hosts documentation for countless open-source Python projects. The hosting service also takes care of rebuilding the documentation when you update your project. Users can browse documentation for every published version, as well as the latest development version.

Sign up at Read the Docs, and import your GitHub repository, using the button *Import a Project*. Read the Docs automatically starts building your documentation. When the build has completed, your documentation will have a public URL like this:

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

The configuration file is named `.readthedocs.yml` in the project directory. 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 the file `docs/requirements.txt`. Note that this file partially duplicates Poetry's lock file. It needs to be kept up-to-date manually, whenever you upgrade Sphinx, and whenever you add, upgrade, or remove a Sphinx extension.

2.17 Tutorials

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

2.17.1 How to test your project

Run the test suite using *Nox*:

```
$ nox -r
```

Additional checks are provided by *pre-commit*:

```
$ pre-commit run --all-files
```

2.17.2 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>
```

2.17.3 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.

2.17.4 How to push code changes

Create a branch for your changes:

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

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*.
- *Coverage data is uploaded to Codecov*.

2.17.5 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.

Opening the pull request triggers another automated step:

- Codecov comments on the pull request, summarizing how the changes impact code coverage.

2.17.6 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 master branch*.
- *Coverage data is uploaded to Codecov*.
- *The draft GitHub Release is updated*.
- *A pre-release of the package is uploaded to TestPyPI*.

In your local repository, update the master branch:

```
$ git switch master
$ git pull origin master
```

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).

2.17.7 How to make a release

Before making a release, go through the following checklist:

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

Making a release is a two-step process:

1. Bump the version using poetry version. (Commit and push.)
2. Publish a GitHub Release.

When bumping the version, adhere to Semantic Versioning and PEP 440. The individual steps for bumping the version are:

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

If you want the Git tag to be annotated or signed, add the following optional steps:

```
$ git tag --message="<project> <version>" v<version>
$ git push origin v<version>
```

To publish the release, locate the draft release on the *Releases* tab of the GitHub repository, and follow these steps:

1. Click **Edit** next to the draft release.
2. Enter a tag of the form `v<version>`, using the new project version.
3. Enter the release title, e.g. `<version>`.
4. Edit the release description, if required.
5. Click **Publish Release**.

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

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

Update your local repository:

```
$ git switch master
$ git pull origin master v<version>
```

2.18 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 3.6+ and the following tools:

- Cookiecutter
- Poetry
- Nox

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.

You can ensure that your changes adhere to the code style by reformatting with Black:

```
$ nox --session=black
```

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`)
- Build System and Dependencies (`build`)

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 master 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, 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 2.0, available at https://www.contributor-covenant.org/version/2/0/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

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

CHAPTER

SIX

USAGE

```
$ cookiecutter gh:cjolowicz/cookiecutter-hypermodern-python \
  --checkout="2020.4.15"
```

CHAPTER

SEVEN

FEATURES

- Packaging and dependency management with Poetry
- Test automation with Nox
- Continuous integration with GitHub Actions
- Documentation with Sphinx and Read the Docs
- Automated uploads to PyPI and TestPyPI
- Automated release notes with Release Drafter
- Code formatting with Black and Prettier
- Testing with pytest
- Code coverage with Coverage.py
- Coverage reporting with Codecov
- Command-line interface with Click
- Linting with Flake8 and various *awesome plugins*
- Static type-checking with mypy and pytype
- Runtime type-checking with Typeguard
- Security audit with Bandit and Safety
- Git hook management with pre-commit
- Checked documentation examples with xdoctest
- API documentation with autodoc, napoleon, and sphinx-autodoc-typehints

The template supports Python 3.6, 3.7, and 3.8.

CHAPTER

EIGHT

FAQ

What is this project about?

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

What makes this project different from other Python templates?

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

Our goals are:

- Keep a 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

Why is this Python template called "hypermodern"?

Hypermodernism is a school of chess from 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 outdated visions of the future.