Python Packaging Essentials

Joseph Barbier

2025-06-21

Table of contents

Preface

The aim of this site it to provide all the must known practices when it comes to create a Python package. It offers **multiple blog posts**, where each of them covers one topic with a few key points. The goal here is to empower anyone with just basic Python knowledge.

We'll go over concrete examples, use clear explanations, and try as much as possible to go straight to the point so that anyone with some Python knowledge can create their own Python packages.

Good to know

- All blog posts are **independant**. Even if they follow some sort of order, it's perfectly fine to just look at what interests you.
- This site is open source and in **continuous improvement**. If you want to suggest an improvement (correct an error, improve an explanation, add an example or anything else), it's more than welcomed. It starts here.
- You can download the **PDF** version of this site in the top left of the site / navigation bar.
- This website is **not exhaustive** and does not aim to be. We aim to focus on the key points and refer to other resources for more specific topics.

Contributing

Set up environment

In order to follow the steps below, you'll need to have both Git, uv and Quarto installed on your machine.

- Fork the Github repo
- Git clone it:

```
git clone https://github.com/YourUsername/python-packaging-essentials.git
```

- Create a new Git branch

```
git checkout -b branch-name
```

- Set up your environment

0.0.0.1 MacOS/Unix-like

```
uv python install
uv sync
source .venv/bin/activate
```

0.0.0.2 Windows

```
uv python install
uv sync
.venv\Scripts\activate
```

Make changes

Each blog post lives in a Quarto file (`.qmd`). It's a mix of markdown and chunk of code.

You can preview locally your changes with:

```
quarto preview
```

If not done automatically, open your browser at http://localhost:4000/

Part I

Create a package

1 Introduction to packaging

Creating a Python package is all about making your code **reusable**, **shareable**, and **easy to install**. Whether you want to publish a library for the world or just organize your own projects better, understanding how packaging works is the first step.

1.1 The `__init__.py` File

At its simplest, a package is just a folder that contains an `__init__.py` file.

```
my_package/
   __init__.py
```

The presence of `__init__.py` tells Python: "this is a package."

Even if it's an empty file, it's important—it allows you to import parts of your code like this:

```
from my_package import something
```

Without it, Python treats the folder as a regular directory, **not something it can import from**.

1.2 Package vs Module

These terms get thrown around a lot. Here's the quick breakdown:

- **Module**: A single `.py` file (e.g., `my_file.py`)
- **Package**: A directory with an `__init__.py`, possibly containing multiple modules (e.g., multiple files)

1.3 PyPI (Python Package Index)

PyPI is the default online repository for Python packages. This is where packages are stored so that others can find and install them.

When you run:

```
pip install requests
```

You're downloading the `requests` package from PyPI. More specifically, you're downloading the package's distribution (which might be source code or precompiled binaries) to your local machine from PyPI servers.

1.4 pip (and friends)

`pip` is the tool used to install packages from PyPI. It's simple and widely supported.

Example:

```
pip install numpy
```

But when working with packages in Python, you need to take into account the package version. Maybe you need `numpy 2.1.2` instead of `2.1.1` for your project.

You can read more about this in the handling dependencies article, but in summary, it's important to control the version of the packages you use/distribute, to ensure reproducible workflows and avoid unexpected things.

Some newer tools are built around `pip` to offer additional features such as **dependency management** and **better performance**.

One of the most important things these tools do is called dependency resolution, which involves calculating which versions of each package are compatible with each other based on version constraints. For example, you might be using a version of numpy that is incompatible (for whatever reason) with matplotlib, and since matplotlib relies on numpy, there's a problem.

Since 2024, the best tool available is called uv. It's super easy to use, super fast and does everything you need, in one place, with one tool. It's more of a Python project manager than a simple package installer.

2 Organize a package

In order to follow the steps below, you'll need to have both Git and uv installed on your machine. Both are command-line tools, meaning you'll use your terminal to run commands that perform various actions.

Let's assume we're naming our Python package *"sunflower"*.

2.1 Initialize the directory

The very first step is to create a new directory named *"sunflower"*. Inside this directory, create another directory with the same name. The structure should look like this:

```
sunflower/
   sunflower/
```

2.2 Add main Python project files

Next, we need to create a few essential files at the root of the project.

2.2.1 `pyproject.toml`

All the package metadata. It will contain a lot of useful information when we want to distribute this PyPI package so that everyone can install it easily.

Here is a simple version of this file:

```toml
[project]
name = "sunflower"
version = "0.1.0"
description = "create pretty sunflowers"
license = "MIT"
license-files = ["LICENSE"]
keywords = ["sunflower", "flower"]
```

```
authors = [
    { name = "Your Name", email = "yourmail@example.com" },
]
readme = "README.md"
requires-python = ">=3.9"
dependencies = ["matplotlib"]

[build-system]
requires = [
    "setuptools",
    "setuptools-scm",
]
build-backend = "setuptools.build_meta"

[tool.setuptools]
packages = ["sunflower"]

[tool.uv.sources]
sunflower = { workspace = true }

[project.urls]
Homepage = "https://sunflower-are-beautiful.com"
Issues = "https://github.com/YourName/sunflower/issues"
Repository = "https://github.com/YourName/sunflower"
```

2.2.2 LICENSE

A basic text file containing the licence for your package. This licence is important because it tells other people what they are allowed to do with your package.

It is specific to each project, but you can find out more at choosealicence.com.

Here is an example of the most common licence: the MIT licence.

```
Copyright (c) 2025 Your Name

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.
```

2.2.3 `.git/`

This is a directory used internally by the Git software to track all changes in the project. Assuming that the first *"sunflower"* directory is in your `Desktop/` directory, you should create this directory by running the `git init` command when you are in the `Desktop/sunflower/` directory.

It's very likely that **you won't see it**, as most operating systems (Windows, MacOS, etc.) hide files/directories that start with `.`, but it doesn't matter. This directory will be managed entirely by Git itself, so we recommend that you **never make any manual changes to it**.

2.2.4 `.gitignore`

A file in which each line describes one or more files/directories that are not explicitly part of the project or are not relevant in general. Don't worry too much about this, you can just start with the example content below.

It is very likely that **you will not see it** outside your code editor, as most operating systems (Windows, MacOS, etc.) hide files/directories that start with `.`, but that doesn't matter.

```
# Python-generated files
__pycache__/
*.py[oc]
build/
dist/
wheels/
*.egg-info

# Virtual environments
.venv/
```

```
venv/
.env/
env/

# VS code config
.vscode/

# files on mac
.DS_Store

# all cache files
*cache*

# Sandbox files
sandbox.py
sandbox.ipynb
```

2.2.5 `.venv/`

A directory containing all the things we need to work properly in our Python environment. It contains a Python interpreter, all the packages used in the project (e.g. `numpy`, `requests`, etc), and a few other things.

The best way to create one is to run `uv venv`.

It is very likely that you will **not** see it outside your code editor, as most operating systems (Windows, MacOS, etc.) hide files/directories that start with `.`, but that doesn't matter.

2.2.6 `README.md`

A markdown file that describes the project, gives advice on how to use it, install it and so on. There are no rules about what to do with this file, it's just used to tell people what is the first thing they should read before using your package.

For example, it could be something like this:

```
# sunflower: my cool Python package

Welcome to the homepage of the `sunflower` project.

It's a new project, but it will be available soon!
```

2.2.7 `sandbox.py`

A file that we will use to test and use our package. It's optional but very practical.

As you can see, we haven't written a single line of Python code, yet we already have a lot of files and directories. The organisation of our project now looks like this:

```
sunflower/
   sunflower/
   .git/
   .venv/
   .gitignore
   README.md
   LICENSE
   sandbox.py
   pyproject.toml
```

2.3 Add Python code

When creating a Python package, we want to write **a reusable piece of code**, not just put in a few scripts that do things. To illustrate:

2.3.1 Not reusable code

```
name = "Joseph"
message = f"Hello {name}"
print(message)
```

```
Hello Joseph
```

This code does something: it prints a message.

2.3.2 Reusable code

```
def say_hello(name):
   message = f"Hello {name}"
   print(message)
```

The code above does 'nothing'. The only thing it does is create a function object that will be stored in memory. I can now call it and it will execute some code. For example:

```
say_hello("Joseph")
```

```
Hello Joseph
```

Here, we'll keep things simple and assume that we only want to provide functions and classes in our package.

Now, let's create our first Python module (which is just a file ending in `.py`). We'll call it `module1.py`, but it can be anything. The only thing we want to stick to when naming files is:

- use lowercase only
- avoid spaces and odd characters
- keep it short
- use underscores "_"

2.3.3 Bad file names

```
my file.py
Myfile.py
myFile.py
my@file.py
my-file.py
this-file-does-this-and-that.py
```

2.3.4 Good file names

```
my_file.py
myfile.py
```

So let's put our `module1.py` in `sunflower/sunflower/`, which will give us:

```
sunflower/
   sunflower/
      module1.py
   .git/
   .venv/
   .gitignore
   README.md
   LICENSE
   sandbox.py
   pyproject.toml
```

In `module1.py`, we're going to add our very first function to our package. For example, we'll create a `count_sunflower()` function. This function will count how many times the word 'sunflower' occurs in a given string.

```python
import re

def count_sunflowers(s):
   s = re.sub(r"[^a-zA-Z\s]", "", s)  # Remove non-text characters
   s = s.lower()                      # Convert to lowercase
   n_sunflower = s.split().count("sunflower")
   n_sunflowers = s.split().count("sunflowers")
   return n_sunflower + n_sunflowers
```

We're now going to add a `__init__.py` file in the same place as the `module1.py` file that contains our previous function. This is a special Python file. It tells Python that the `sunflower/sunflower/` directory is a package, which will allow us to import functions from that package into the outside world.

> Learn more the `__init__.py` file here.

We now have this:

```
sunflower/
   sunflower/
      __init__.py
      module1.py
   .git/
   .venv/
   .gitignore
   README.md
   LICENSE
```

```
   sandbox.py
   pyproject.toml
```

The `__init__.py` file should look like this:

```
from .module1 import count_sunflowers

__all__ = ["count_sunflowers"]
```

And well done! You may not have realised it, but we already have a Python package that can be used with a function.

2.4 User perspective

Let's now look at how to use our package from the user's point of view.

Once again, we'll need to run a command in our terminal at `Desktop/sunflower/`:

```
uv pip install -e .
```

This command will install our current package in editable mode. This allows us to test our package while making updates.

The next step is to open `sandbox.py` and write some code that uses our package.

```
from sunflower import count_sunflower

text = """
Sunflower petals bright and gold,
Sunflower fields, a sight to behold.
Sunflower dreams in the morning light,
Blooming softly, pure and bright.
"""

print(count_sunflower(text))
```

3

Note that we can use any of the following syntaxes:

2.4.1 Syntax 1

```
from sunflower import count_sunflower

text = """
Sunflower petals bright and gold,
Sunflower fields, a sight to behold.
Sunflower dreams in the morning light,
Blooming softly, pure and bright.
"""

count_sunflower(text)
```

2.4.2 Syntax 2

```
import sunflower

text = """
Sunflower petals bright and gold,
Sunflower fields, a sight to behold.
Sunflower dreams in the morning light,
Blooming softly, pure and bright.
"""

sunflower.count_sunflower(text)
```

2.4.3 Syntax 3

```
import sunflower as sfl # or any other alias like "sf" or "sunflo"

text = """
Sunflower petals bright and gold,
Sunflower fields, a sight to behold.
Sunflower dreams in the morning light,
Blooming softly, pure and bright.
"""

sfl.count_sunflower(text)
```

You can learn more about it in the API design blog post.

2.5 Add an internal function

When creating a package, it is very practical to create functions that we will use internally: inside the package itself.

If we go back to our previous example, we might want to have a separate function that takes a string and cleans it up by removing non-text characters and putting it in lower case. Let's name this function `_clean_string()` and place it in a new file: `module2.py`.

```
import re

def _clean_string(s):
    s = re.sub(r"[^a-zA-Z\s]", "", s)  # Remove non-text characters
    s = s.lower()                      # Convert to lowercase
    return s
```

Our code in `module1.py` should now become:

```
from .module2 import _clean_string

def count_sunflowers(s):
    s = _clean_string(s)
    n_sunflower = s.split().count("sunflower")
    n_sunflowers = s.split().count("sunflowers")
    return n_sunflower + n_sunflowers
```

We now have 2 functions:

- `count_sunflowers()` a **public** function that users of the package will use.
- `_clean_string()` a **private** function used internally. The underscore ('_') at the beginning of the function name tells other people that it should not be used outside the package from which it came.

Note that `_clean_string()` is still usable by users if they run it:

```
from sunflower.module2 import _clean_string
```

But as you can see from the documentation blog post, we won't have or create documentation on these functions, so they're unlikely to find it anyway.

2.6 Final organization

After all these steps, our package now looks like this:

```
sunflower/
   sunflower/
       __init__.py
       module1.py
       module2.py
   .git/
   .venv/
   .gitignore
   README.md
   LICENSE
   sandbox.py
   pyproject.toml
```

3 Handling dependencies

Dependencies are the external Python packages your code needs in order to work—like `requests`, `numpy`, or `pandas`.

Here we'll focus on using uv to handle dependencies, as it's currently the best tool for this out there (it's fast and fairly easy to use, especially if you know `pip`).

3.1 Specify dependencies

For example, let's say we have this function in our package:

```
import numpy as np

def normalize(array):
   min_val = np.min(array)
   max_val = np.max(array)
   return (array - min_val) / (max_val - min_val)
```

When people want to use our function, they **need** to have `numpy` installed for it to work, otherwise it will raise a `ModuleNotFoundError` on their machine.

So in order to ensure they get `numpy` installed, we set `numpy` as a dependency of our package. The dependencies of a package are listed in the `pyproject.toml` file. If you don't know what that is, check out organizing a package.

With uv, we just have to run:

```
uv add numpy
```

This will automatically add `numpy` to our `pyproject.toml`.

3.1.1 Before

```toml
[project]
name = "mypackage"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = []
```

3.1.2 After

```toml
[project]
name = "mypackage"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
    "numpy>=2.2.4",
]
```

If we add other dependencies, they will be added to the `dependencies` list.

3.2 Avoiding dependencies

In general, we want to **avoid** having too many dependencies. Why is that? Because when we install a package, we need to install its dependencies too, as well as the dependencies of those packages, and so on.

The issue with this is that it adds a lot complexity quickly and increase the risk of having **conflicts**.

For example, one package might need a version of `numpy` before `<2.0.0`, while another need a version above or equal to `>=2.2.0`. This kind of situation can quickly arise if not careful when adding too many dependencies, and it's usually a nightmare to resolve.

> Note: packages with low or no dependencies are called **lightweight**. As an example, have a look at the narwhals package.

The thing with having lots of dependencies is that it makes it easier for you to write code because you can use other people code super easily. So it's always a trade-off somehow.

Always ask yourself those questions before adding a new package to your dependencies:

- is the dependency a well-known, stable package (`numpy`, `requests`, etc) or is it new and is likely to change in the future?
- does this dependency has lots of dependencies too? This might be a red flag
- can't you just code what it does yourself? If you only need a single function, go check their source code on Github and see if it's easy to do on your side (and ensure their License allow you to copy the code too).

3.3 Controlling the versions

What's the difference between `numpy 2.1.0` and `numpy 2.0.0`? Well, many things, but for example, in `numpy 2.0.0`, the `np.unstack()` function doesn't exist as it's a new one from `numpy 2.1.0`.

If our package relies on `np.unstack()` in one of our functions, we **can't** let people install any `numpy` version when installing our package. We need to ensure people install this version: `numpy>=2.1.0`. If we translate it, it means *any version of `numpy` above or equal to `2.1.0`*. Let's see some other examples.

3.3.1 `numpy==2.1.0`

Install exactly this version of `numpy`.

```
uv pip install numpy==2.1.0
```

3.3.2 `numpy<=2.1.0`

Install the latest available version before `2.1.0` (including `2.1.0`) of `numpy`.

```
uv pip install 'numpy<=2.1.0'
```

3.3.3 `numpy>=2.1.0,<2.2.0`

Install the latest version between `2.1.0` (included) and `2.2.0` (excluded) of `numpy`.

```
uv pip install 'numpy>=2.1.0,<2.2.0'
```

3.3.4 `numpy`

Install the latest version of `numpy`.

```
uv pip install numpy
```

Note that for each of those, the package resolver will always try to install the latest version it can depending on the other dependencies. If a package requires `numpy<=2.1.0`, other packages **must** include `numpy 2.1.0` for it to work.

At this point, you might ask, **how do I know** which versions of each dependencies are required for my package? Well, as far as I know, there is no easy answer to this, but there are ways to ensure you don't get unexpected behaviors.

3.3.5 Set the minimum version required...

For each of your dependencies, set in your `pyproject.toml` the minimum version required. With uv, you can run the following to install a specific version:

```
uv pip install numpy==2.0.0
```

> Warning: the command above will install a specific version of `numpy`, but will not change the requirements in `pyproject.toml`.

3.3.6 ... test your code...

You have to test that your code works as expected on those versions. The best way to do that is **unit testing**, and it's the point of the next blog post.

3.3.7 ... and be convenient

> Dependening on whether you're planning on distributing your package (e.g., put it on PyPI and allow other people to install it) or not, you might want to do different things here. We'll assume you want to distribute it at the end.

When your package goaled is to be installed by other people, you want to be convenient. By that I mean **not being too restrictive**.

If we take our example from before, we know that we need at least `numpy==2.1.0` for our package to work, but we also know that **any numpy version above works too**. For this reason, we set `numpy>=2.1.0` instead of `numpy==2.1.0` to allow a broader range of possibility.

3.4 Breaking changes

By default, when installing our package, it will try to find the latest `numpy` version that satisfies the requirements.

But, you might say there's a risk it will break on a new `numpy` version? Yes, **it absolutely does**. And that's exactly why we said earlier why we wanted to avoid having too many dependencies and use stable ones only.

The good thing with packages like `numpy` is that it's one of the most important Python package and one of its core component. They **can't** make breaking changes on any significant feature. When they want to do it, they usually add warnings like this: *"The function xxx is deprecated and will be removed in a future version, please use yyy instead?"*.

But, if you want to be sure you don't get breaking changes, set the maximum version of the dependencies, with things like `numpy<=2.2.0`. This will ensure it's safe, but this also means you'll need to manually update it as new versions come out.

3.5 Required, Optional and Dev dependencies

When working with dependencies, it's useful to differentiate between three main types: required, optional, and development dependencies. Each serves a different purpose in your package.

3.5.1 Required

Required dependencies are the ones we've been discussing so far - packages that your code absolutely needs to function properly. These go in the `dependencies` list in your `pyproject.toml`.

```toml
[project]
dependencies = [
    "numpy>=2.1.0",
    "pandas>=2.0.0",
]
```

3.5.2 Optional dependencies

Optional dependencies are packages that enhance your code but aren't strictly necessary for core functionality. For example, if your data processing package works with CSV files by default but can also handle Excel files with an additional dependency.

You can specify these in your `pyproject.toml` using the `[project.optional-dependencies]` section:

```toml
[project.optional-dependencies]
excel = ["openpyxl>=3.1.0"]
plot = ["matplotlib>=3.7.0", "plotly>=5.23.0"]
```

This lets users install only what they need:

3.5.2.1 `mypackage`

Install your package required dependencies **only**:

```
uv pip install mypackage
```

3.5.2.2 `mypackage[excel]`

Install your package required dependencies as well `openpyxl`:

```
uv pip install "mypackage[excel]"
```

3.5.2.3 `mypackage[plot]`

Install your package required dependencies as well `matplotlib` and `plotly`:

```
uv pip install "mypackage[plot]"
```

3.5.2.4 `mypackage[excel,plot]`

Install with all optional dependencies:

```
uv pip install "mypackage[excel,plot]"
```

In your code, you'll need to handle cases where optional dependencies aren't installed:

```
def read_file(filename):
    if filename.endswith('.csv'):
        # Core functionality
        import pandas as pd
        return pd.read_csv(filename)
    elif filename.endswith('.xlsx'):
        try:
            # Optional functionality
            import openpyxl
            import pandas as pd
            return pd.read_excel(filename)
        except ImportError:
            raise ImportError(
                "Excel support requires 'openpyxl'. "
                "Install with 'pip install mypackage[excel]'"
            )
```

This will give your users a **clear and meaningful error message** that they can resolve very quickly. This kind of thing exist for the same reason we're talking about in this article: trying to minimize the number of dependencies (especially the unused ones!).

In order to add package to your optional dependencies, you can run:

```
uv add matplotlib --optional plot
```

This will add `matplotlib` to the `plot` section in the optional dependencies in your `pyproject.toml`.

3.5.3 Dev dependencies

Development dependencies are packages you need only when developing your package, not when using it. These include testing frameworks, documentation generators, linters, and similar tools.

Specify these in your `pyproject.toml` like this:

```
[project.optional-dependencies]
dev = [
    "pytest>=7.0.0",
    "ruff>=0.11.5",
    "sphinx>=7.0.0",
]
```

In order to add a package to your dev dependencies, you can run:

```
uv add ruff --dev
```

Developers working on your package can install all optional dependencies as well as dev dependencies with:

```
uv sync --all-extras
uv pip install -e .
```

> The `-e` flag installs your package in "editable" mode, meaning changes to your code take effect immediately without needing to reinstall. This might not be necessary as `uv` is already doing it in the first command, but let's explicit it here.

Part II

Code quality

4 Unit tests

This page is a work in progress. You can see the current state of the project here.

5 Writing documentation

This page is a work in progress. You can see the current state of the project here.

6 Errors and warnings

This page is a work in progress. You can see the current state of the project here.

7 API design

This page is a work in progress. You can see the current state of the project here.

Part III

Workflow

8 Github Actions

Having a package implies several things, most importantly:

- creating and deploying a documentation website
- testing that it works as expected using unit tests
- tracking changes with version control (Git)

In this post, we'll walk through 3 essential Github Actions you **need** in your workflow when developing Python packages.

> This blog assumes basic Git/Github knowledge (push/pull, pull requests, branches).

8.1 TLDR: Github Actions

Github Actions are scripts that perform tasks (pretty much anything you want) when specific "events" occur. You can do **a lot** with them, but here we'll focus on practical use cases for developing a Python package.

These scripts live in the `.github/workflows/` directory and are written as `yaml` files. For instance, a Python package named "sunflower" with two different Github Actions might be organized like this:

```
sunflower/
  sunflower/
      __init__.py
      module1.py
      module2.py
  .gitub/
      workflows/
          unit-tests.yaml
          code-format.yaml
  tests/
  .git/
```

```
.venv/
.gitignore
README.md
LICENSE
pyproject.toml
```

On certain "events" (as defined in those scripts), `unit-tests.yaml` and `code-format.yaml` will be triggered.

The events we care about here are:

- Opening a pull request
- Merging or pushing to the main branch

Let's look at a practical example to understand why these scripts are important.

8.2 Unit testing

> If you're not familiar with unit testing, check out this blog post first.

Suppose we have unit tests written with `pytest` in the `tests/` directory. We can now add a `unit-tests.yaml` file in `.github/workflows/` that looks like this:

```yaml
name: Unit tests

on:
  pull_request:
    branches: [main]

jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        python-version: ["3.9", "3.13"]

    env:
      UV_PYTHON: ${{ matrix.python-version }}
    steps:
      - uses: actions/checkout@v4
```

```yaml
      - name: Install uv
        uses: astral-sh/setup-uv@v5

      - name: Enable caching
        uses: astral-sh/setup-uv@v5
        with:
          enable-cache: true

      - name: Install the project
        run: uv sync --all-extras --dev

      - name: Run tests
        run: uv run pytest
```

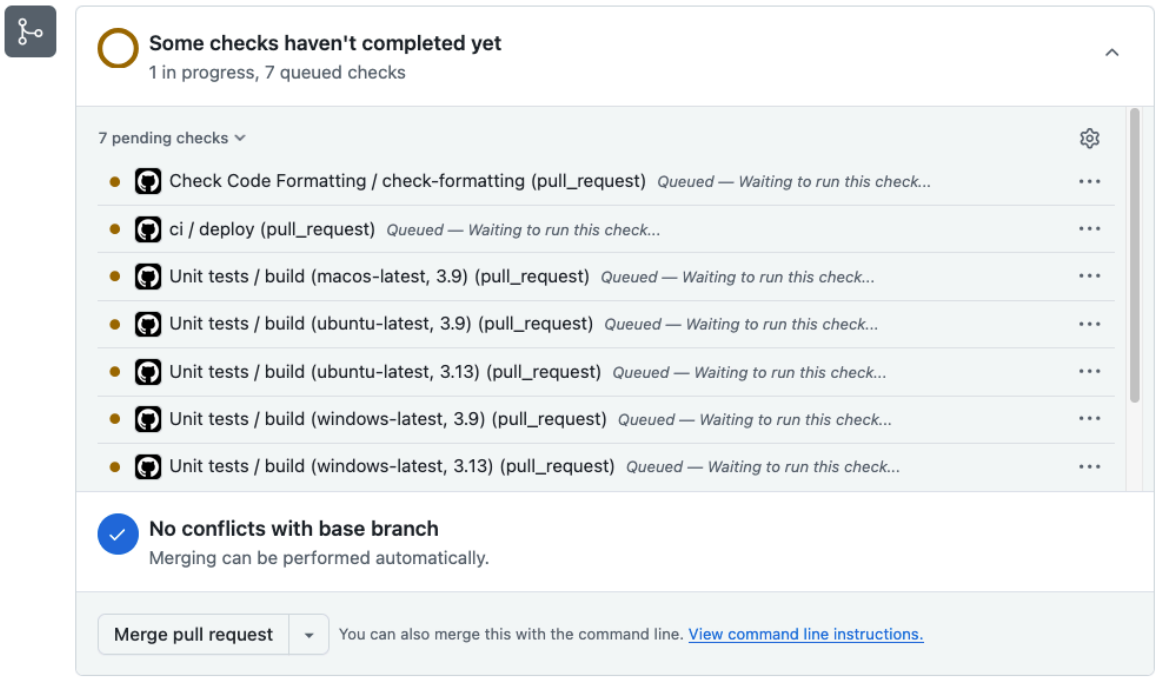
What it does:

When someone pushes a commit to a branch with an open pull request (as specified in the `on` section), this action will:

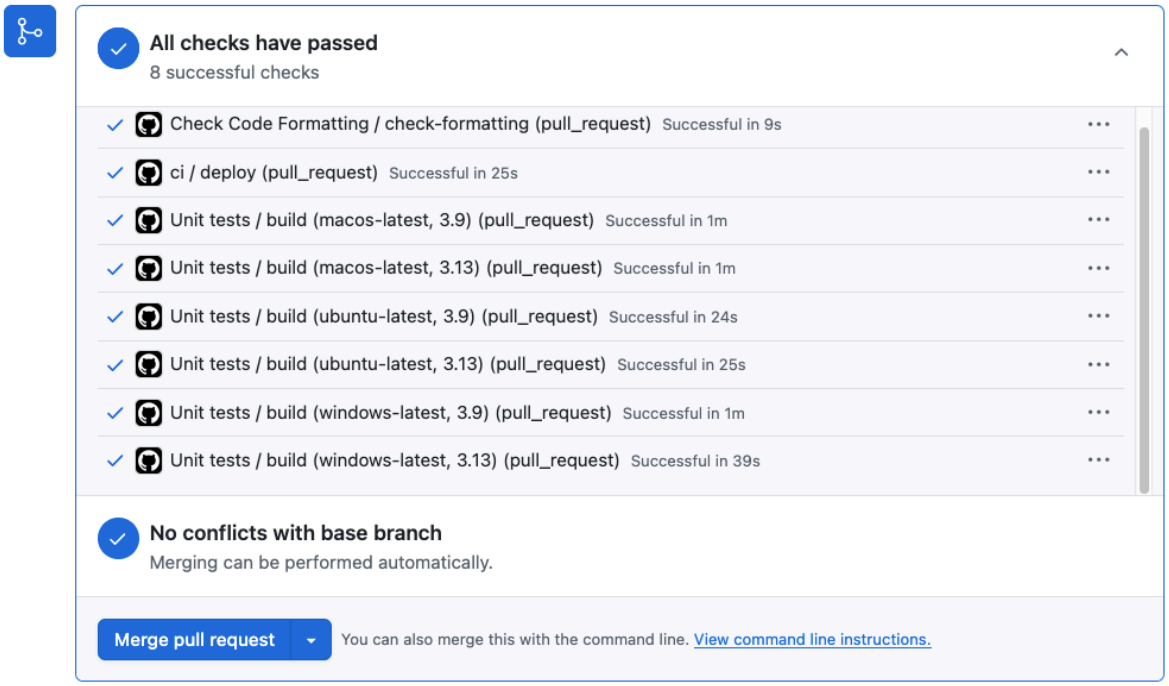
- install uv (a Python package manager)
- install the project dependencies using `uv sync --all-extras --dev`
- run the test suite with `uv run pytest`

This runs across multiple Python versions (3.9 and 3.13) and operating systems (Windows, macOS, and Linux). That gives us 6 combinations in total (2 Python versions × 3 OSes).

Here's what shows up on the pull request while the tests are running:



If any of those combinations fail (meaning at least one test fails), you'll see a message indicating that something didn't work. For example, you might learn that your package fails on Windows with Python 3.9. Otherwise, you'll see something like this:



The purpose of setting up this Github Action is to **automatically and easily** verify that the package works in different environments, helping ensure that only valid code is merged into the main branch.

In this example, we used just two Python versions and one set of dependencies, but this approach can be extended to test the package under many more scenarios. That way, we get a clear and precise picture of what works and what doesn't.

8.3 Create and deploy documentation

> There's a dedicated blog post on generating and deploying documentation for your package. Check it out here.

Let's say we've created our documentation website with `mkdocs`. We then add a `deploy-site.yaml` file in `.github/workflows/`.

Since generating the documentation website creates a large number of files, it's not ideal to store them in version control. But how do we deploy it to Github Pages if it's not in version control? That's where Github Actions come in!

Now, let's take a look at the following Github Action script:

```yaml
name: ci

on:
  push:
    branches: [main]

permissions:
  contents: write

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Configure Git Credentials
        run: |
          git config user.name github-actions[bot]
          git config user.email 41898282+github-actions[bot]@users.noreply.github.com
      - uses: actions/setup-python@v5
        with:
          python-version: 3.x
      - run: echo "cache_id=$(date --utc '+%V')" >> $GITHUB_ENV
      - uses: actions/cache@v4
        with:
          key: mkdocs-material-${{ env.cache_id }}
          path: .cache
          restore-keys: |
            mkdocs-material-

      - name: Install uv
        uses: astral-sh/setup-uv@v5

      - name: Enable caching
        uses: astral-sh/setup-uv@v5
        with:
          enable-cache: true

      - name: Install the project
        run: uv sync --all-extras --dev

      - name: Deploy MkDocs
        env:
```

```
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: uv run mkdocs gh-deploy --force
```

What it does:

When someone merges or pushes to the main branch, this action will:

- install uv (a Python package manager)
- install the project dependencies using `uv sync --all-extras --dev`
- generate the entire documentation website with `uv run mkdocs gh-deploy --force`
- push the documentation website to the `gh-pages` branch on Github

With this setup, assuming our website is deployed to Github Pages using the `gh-pages` branch, the documentation site is deployed automatically whenever a pull request is opened or we merge/push to the main branch. All without keeping the auto-generated files in version control.

This also **removes all manual work** related to building and deploying the documentation, as it's now fully automated through this Github Action.

8.4 Code linting and formatting

When working on a project, it's crucial to maintain standardized coding practices:

- Consistent formatting (e.g., indentation, spacing, quotes)
- Clean code free of unused imports, bad patterns, or minor bugs

This is where **code formatting** and **linting** tools come into play, and we can automate them using Github Actions.

We'll use `ruff` here, which is a super fast linter and formatter for Python. It can both check for issues (like `flake8` or `pylint`) **and** format code (like `black`), all in one tool.

8.4.1 Add Ruff to your project

First, add Ruff as a development dependency:

```
uv add --dev ruff
```

8.4.2 Create the GitHub Action

Now, create a file named `.github/workflows/code-format.yaml` with the following content:

```yaml
name: Ruff lint and format

on:
  pull_request:
    branches: [main]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install uv
        uses: astral-sh/setup-uv@v5

      - name: Enable caching
        uses: astral-sh/setup-uv@v5
        with:
          enable-cache: true

      - name: Install dependencies
        run: uv sync --all-extras --dev

      - name: Check formatting
        run: uv run ruff format . --check

      - name: Lint code
        run: uv run ruff check .
```

What it does:

When a pull request is opened against `main`, this action will:

- Install your project and its development dependencies using `uv`

- Check if the code is properly formatted with `ruff format . --check`
- Lint the code with `ruff check .` to catch any potential issues

If there's anything wrong (such as a file needing formatting or an unused import), the action will fail, and the pull request will show a red . That's your cue to fix the code.

This ensures that all code added to the codebase is well-formatted and adheres to the established rules.

> Note that there's an additional way to enforce this called `pre-commit`, and there's a dedicated blog post on it.

8.5 FAQ

8.5.1 I didn't understand certain things

Github Actions is one of those things where you need to try it yourself to get the full picture. I recommend creating a basic Python package with documentation and tests, then testing the examples provided to see how they work.

8.5.2 Where does the code run?

When a Github Action is triggered, Github sets up a clean VM (virtual machine) to run your workflow. There are limits on usage, but they're quite generous before you'll need to enter your credit card details.

8.5.3 Can I run it on my machine?

Yes, you can! Thanks to a project called act. In short, it uses Docker to run your Github Actions in the correct context.

9 Pre-commit Hooks

In this post, we'll walk through how to add `pre-commit` to your Python package to enforce good code hygiene *automatically*.

> This post assumes you're already using Git and are familiar with what commits are. It also requires using uv.

9.1 TLDR: What is `pre-commit`?

`pre-commit` is a framework for managing and running "hooks", which are just scripts defined in a `.pre-commit-config.yaml` file that run at specific points in the Git lifecycle. The most common is the `pre-commit` hook, which runs **before** a commit is created.

The point of using pre-commit hooks is to prevent your codebase from including unwanted things, such as unformatted code, oversized files, print statements, and so on.

Here's what makes `pre-commit` awesome:

- **It runs locally** – unlike CI (e.g. GitHub Actions), it catches issues *before* they get pushed
- **It's fast** – runs only on the files you've changed
- **It's customizable** – tons of hooks are available, or you can write your own
- **It integrates with CI** – you can run `pre-commit` in CI to make sure everyone follows the same rules

9.2 How it looks

Here is a pre-commit hook that:

- checks if our code is both linted and formatted
- if not, it will try to fix it

```yaml
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.7
    hooks:
      - id: ruff
        types_or: [python, pyi]
        args: [--fix]
      - id: ruff-format
        types_or: [python, pyi]
```

This will run each time we run `git commit`. If our code is not perfectly linted and formatted, it will prevent the commit and lint/format it (if possible).

Once our code is fixed, we can re-run `git add` and `git commit`, and it will accept our commit, which we can then push.

9.3 How to set up

9.3.1 1. Configuration

Create a `.pre-commit-config.yaml` file at the root of your project.

Let's use a relatively common pre-commit setup. It configures formatting and linting with ruff, checks for large files, and removes trailing whitespace:

```yaml
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.7
    hooks:
      - id: ruff
        types_or: [python, pyi]
        args: [--fix]
      - id: ruff-format
        types_or: [python, pyi]

  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: trailing-whitespace
      - id: check-added-large-files
```

- The `repo` field defines where the actual checks to run are defined
- Each `rev` pin ensures you're using a specific version of that hook for reproducibility
- The `id` defines the exact check to run

9.3.2 2. Install

Install it as a development dependency:

```
uv add --dev pre-commit
uv run pre-commit install
```

If you don't want to add it to your development dependencies, you can simply run :

```
uv pip install pre-commit
uv run pre-commit install
```

9.3.3 3. Commit

Now try editing a Python file (add spaces at the end of the file or change the formatting of something). Then try committing it:

```
git commit -m "Test pre-commit"
```

You'll see the hooks run automatically and fix (or block) your commit if needed.

```
[INFO] Installing environment for https://github.com/astral-sh/ruff-pre-commit.
[INFO] Once installed, this environment will be reused.
[INFO] Running ruff-format...
[INFO] Files were modified by this hook. Please stage the changes and try again.
```

If `pre-commit` fixes files automatically, it will ask you to re-stage them and try committing again. This prevents broken or messy code from slipping into version control.

9.4 FAQ

9.4.1 "Files were modified by this hook"

This means the hook fixed your files. Run:

```
git add .
git commit -m "message"
```

9.4.2 "Hook failed"

Some hooks (like `ruff`) don't fix issues automatically. You'll need to fix them manually based on the error messages.

9.4.3 "Nothing happens when I commit"

Did you run `uv run pre-commit install`? That installs the Git hook. Without it, the hooks won't run.

10 Publish to PyPI

This page is a work in progress. You can see the current state of the project here.