Good practices I

Raul P. Pelaez

February 18, 2026

Contents

1 Introduction

Professional Python development requires more than just writing code, it demands careful organization and reproducibility. Today we focus on creating projects others can understand, use, and extend.

Advice

The usual steps
When confronted with a new project (or starting one yourself), you will always need to follow these high-level steps:

- Get the project's source code
- Install the project's dependencies
- Install the project itself
- Run the project's tests

If the project is already available in pip or conda, the first three steps are combined into a single one, and we can assume the tests have already been run. For instance, installing numpy is as simple as running `pip install numpy` or `conda install numpy`. Otherwise, you will need to follow the four steps above. Typically, the project's README.md file will contain the necessary instructions, and the source code will be available in a repository like GitHub and can be fetched with `git`, which we will cover in the future.
Note that each step involves some "substeps", like creating a virtual environment in order to install the project's dependencies. We will cover these substeps in detail in the following lessons.

Info

Why structure matters

A well-organized project acts like a map:

- Makes your project standardized and thus predictable
- Guides collaborators through your codebase
- Simplifies debugging and maintenance
- Enables reliable dependency management
- Facilitates packaging and distribution

Compare these two approaches for installing a project:

```
# Chaotic approach
python some_random_path/some_script.py  # Might work if dependencies are installed

# Professional approach
conda env create -f environment.yml -n myenv # Ensure dependencies
conda activate myenv
pip install -e .  # Editable install into the environment
pytest # Ensure correctness
```

Advice

The above process works for most Python projects. Sometimes, instead of dependencies being installed with `conda` through an `environment.yml` file, they are installed with pip through a `requirements.txt` file. However, the "gist" of the process remains the same: create an environment with all dependencies, install the project, and run tests.

2 Project anatomy

Info

Basic structure
A minimal Python project template contains:

```
myproject/
|-- mypackage/
|    \-- __init__.py
|    \-- module1.py
|    \-- subpackage/
|        \-- __init__.py
|        \-- helpers.py
|-- tests/
|   \-- test_module1.py
|   \-- test_helpers.py
|-- environment.yml
|-- setup.py
|-- README.md
|-- .gitignore
```

The directory layout prevents accidental imports from the project root. Each `__init__.py` file (even empty) marks a Python package. The `tests` directory contains test files mirroring the package structure. The `environment.yml` file lists dependencies, and `setup.py` makes the project installable. We will cover these in detail later.

Besides these, a number of "dot files" can be present. For instance, `.gitignore` lists files to ignore in version control. These files are used to guide/customize the behavior of tools like git, pytest, and others.
Examples:

- SciPy: Scientific computing library
- TorchMD-Net: Machine learning for molecular dynamics
- Home Assistant: Home automation platform

Advice

The details of the project structure can vary. For instance, instead of a `src` directory, some projects use a `scripts` directory (or any other name) with the package inside. Or you might find a `requirements.txt` (pip) file instead of an `environment.yml` (conda) file for dependencies. Or a `pyproject.toml` file instead of a `setup.py` file for packaging. But the general idea remains the same: a clear structure that separates code from tests and dependencies and clearly lists dependencies and installation instructions.

Warning

Common anti-pattern
Avoid placing modules directly in the project root:

```
project/
\-- module1.py
\-- script.py
```

This leads to import issues and naming conflicts. Always use a subdirectory (like `src`) for your packages.

2.1 The `__init__.py` file

Info

`__init.py__`
When present in a directory, this file means "treat this directory as a Python package". It can be empty, but it is often used to:

- Import submodules
- Define package-level variables

Example:
Assuming the structure from before:

```
# mypackage/__init__.py
from .module1 import func1
from .subpackage.helpers import func2
MY_CONSTANT = 42
```

Now, from another script, we can do:

```
from mypackage import func1, func2, MY_CONSTANT
```

The import line automatically loads the `__init__.py` file located under the `mypackage` folder, making the functions and constants available.

Advanced

Dunder names: The __ prefix/suffix in Python
In Python, names surrounded by double underscores (like `__init__.py`) are special names. They are called "dunder" names (short for "double underscore"). These names have a special meaning in the language and are reserved for specific purposes. For instance, `__init__.py` is used to mark a directory as a Python package.

2.2 Dependency management

Each project has its own dependencies (i.e. external libraries/tools it relies on). The community has developed tools, such as `conda` and `pip`, to manage these dependencies in an automated and reproducible way.

Info

Conda environments: The environment.yml file
Instead of manually installing packages, we use conda environment files in projects that list all dependencies. Create reproducible environments with `environment.yml`:

```
name: analysis-env
channels:
  - conda-forge
dependencies:
  - python>=3.9
  - numpy
  - pandas
  - pip
  - pip: # Packages that will be installed with pip
    - matplotlib
```

We can list specific versions for each package to ensure compatibility across installations. The `pip` section allows installing packages not available in conda. When this file is present, create the environment with:

```
conda env create -f environment.yml
```

A new environment named `analysis-env` is created with all dependencies. Activate it with:

```
conda activate analysis-env
```

Info

YAML
YAML is a human-readable data serialization format similar to JSON. It is commonly used for configuration files and data exchange.

Info

Pip and conda
`pip` is the default package manager for Python. It installs packages from the Python Package Index (PyPI). `conda` is a package manager for any software (not just Python) and is part of the Anaconda distribution. It installs packages from the Anaconda repository. It just so happens that conda and its repositories also provide most Python packages.
While `conda` and `pip` kind of get along (see for instance how an `environment.yml` file can list packages to be installed with `pip`), it is generally recommended to stick to one of them for a project. Always try to install packages with `conda`, and only use `pip` when that fails.
Additionally, packages installed via `pip` get installed to the current `conda` environment. Even when using `pip` to install packages, it is recommended to do so from within a `conda` environment.
`pip` also has a file similar to `environment.yml` called `requirements.txt` that allows to list the dependencies of a project, it has a similar syntax:

```
numpy==1.21
pandas==1.3
pytest
```

2.3 Packaging projects

Info

Making your project pip-installable: setup.py
Make your project installable with a minimal `setup.py`:

```
from setuptools import setup, find_packages

setup(
    name="mypackage",
    version="0.1.0",
    packages=find_packages(),
)
```

This file, that should be placed in the projects root folder, assumes the project structure shown earlier. The `find_packages` function locates packages (any folder with a `__init__.py` file). Many other options exist to customize the installation, such as defining entry points for terminal commands, and more. This enables two crucial commands:

```
pip install .          # Regular install
pip install -e .       # Editable (development) mode
```

Regular install copies all necessary files to the Python environment. Editable mode creates a symlink to the project directory, allowing changes to take effect immediately.

Advanced

Modern packaging with pyproject.toml

New projects should prefer `pyproject.toml` over `setup.py`:

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "mypackage"
version = "0.1.0"
requires-python = ">=3.9"
dependencies = [
    "numpy>=1.21",
    "pandas>=1.3"
]
```

Benefits:

- Single configuration file
- No executable setup.py
- Compatible with pip and build

Advice

One rarely writes a `setup.py` file from scratch

In my experience, it is rare to write a `setup.py` file from scratch. I normally copy an existing one (say, from a previous project) and modify it to suit the new project. ChatGPT is also kind of decent at generating a `setup.py` file if you provide it with the necessary information.

For instance, go to the root of your project and run the `tree` command:

```
tree
```

This will show you the directory structure of your project. Copy paste the output to Chat and ask it to generate a `setup.py` file for you.

Advice

A small reminder: Defensive programming

Consider these function improvements:

```python
# Before
def process(data, threshold=0.5):
    results = []
    for item in data:
        if item > threshold:
            results.append(item * 2)
    return results

# After
from typing import List, Iterable

def process_data(
    data: Iterable[float],
    threshold: float = 0.5
) -> List[float]:
    """Filter and transform values above threshold."""
    if not isinstance(threshold, (int, float)):
        raise TypeError("Threshold must be numeric")

    return [item * 2 for item in data if item > threshold]
```

Improvements include:

- Type hints for clarity
- Input validation
- Docstring explanation
- Faster and simplified list comprehension

2.4 What this project about?: The README.md file

The `README.md` file is the first thing people see (and look for) when visiting your project. It should contain, at least:

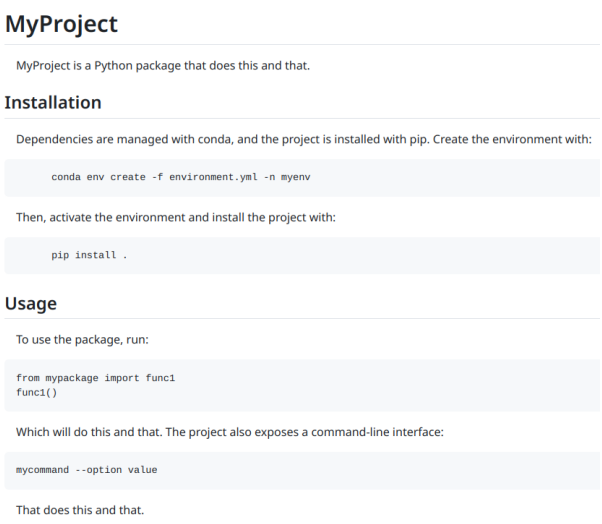
- A brief description of the project
- Installation instructions
- Usage examples

Example:

````
# MyProject

MyProject is a Python package that does this and that.

## Installation

Dependencies are managed with conda, and the project is installed with pip.
Create the environment with:
```bash
conda env create -f environment.yml -n myenv
```
Then, activate the environment and install the project with:
```bash
pip install .
```
## Usage

To use the library from Python use:
````

````
```python
from mypackage import func1
func1()
```
Which will do this and that.
The project also exposes a command-line interface:
```bash
mycommand --option value
```
That does this and that.
````

This file will be rendered if available by platforms like GitHub. The above example will result in this image:



MyProject

MyProject is a Python package that does this and that.

Installation

Dependencies are managed with conda, and the project is installed with pip. Create the environment with:

```
conda env create -f environment.yml -n myenv
```

Then, activate the environment and install the project with:

```
pip install .
```

Usage

To use the package, run:

```
from mypackage import func1
func1()
```

Which will do this and that. The project also exposes a command-line interface:

```
mycommand --option value
```

That does this and that.

Info

Markdown: the .md extension
Markdown is a lightweight markup language with plain text formatting syntax. It is often used to format readme files, for writing messages in online discussion forums, and to create rich text using a plain text editor. Chat is really good with it.

3 Exercises

3.1 Your first project

Goal

Lets create a simple project that we can install and run. You are encouraged to customize the details, but I propose to you the following project: An application that will list the closest asteroids to Earth around a given date. The application will use the NASA API to fetch the data and display it in a user-friendly way when called.
After you are done, you should be able to run the following command in your terminal, with a similar output:

```
$ asteroids --date 2025-01-29
During the last 7 days, the closest asteroids to Earth were:
- Object (2020 YJ2), 45m diameter: 0.1 AU from Earth
- Object (2019 YE), 10m diameter: 0.2 AU  from Earth
- Object (2020 YJ2), 15m diameter: 0.23 AU from Earth
```

Advice

Consider the asteroid thing just a proposal, feel free to change the details around, for instance using the API that you chose for last weeks exercises.

Milestone

Create the project structure, we will use the following:

```
asteroids/
|-- asteroids/
|    \-- __init__.py
|    \-- cli.py
|    \-- api.py
|-- tests/
|   \-- test_cli.py
|   \-- test_api.py
|-- environment.yml
|-- setup.py
|-- README.md
```

Create all the necessary folders and open all the files in VSCode.

hint

- You will need to use the usual shell commands: `mkdir` to create folders, `cd` to move around, `ls` to look, `code` to edit files.

Milestone

Create the `environment.yml` file with the necessary dependencies. You will need to list `requests`, `pandas` and `pytest`. Give the environment a name in the file, create it and activate it.

Milestone

For the moment, write a function called `main` in the `cli.py` file that prints "Hello, world!" when called. We will make this function a terminal command later.

Milestone

Create the `setup.py` file. You can use the example from the lesson. We will write a function inside the `cli.py` called `main` that we want to become a terminal command, so we will need to use the `entry_points` argument in the `setup.py` file. Add this argument to `setup` in the `setup.py` file:

```
entry_points={
    "console_scripts": [
        "asteroids=asteroids.cli:main",
    ],
},
```

Make sure you can install the project by running `pip install .`. If all went well, you should be able to run `asteroids` in the terminal and see "Hello, world!" printed.

Milestone

Write a function in the `api.py` file that fetches the data from the NASA API. You can use the `requests` library for this. Essentially, copy the functionality from the last weeks exercises. Create a function that receives a date and returns the closest asteroids to Earth around that date as a dictionary. For instance:

```
def get_asteroids(date: str) -> dict:
  """Get a list of asteroids that have approached Earth around a given date.

  Args:
          date (str): Date to search for asteroids (YYYY-MM-DD)

  Returns:
          dict: List of asteroids with their name, distance from Earth
                in km, diameter in meters, and date of approach
  """
```

hint

You can start by writing a mock version that just returns a hardcoded dictionary regardless of the date, and build up from there.

Milestone

Modify the `main` function in `cli.py` to call the `get_asteroids` function and print the results in a pretty format. Add an command-line argument to the `main` function that receives the date to search for.

hint

- Use `argparse` to give `main` the ability to receive arguments from the terminal.
- You can import a file from within the same folder by writing `from .api import get_asteroids`. Notice the dot before the module name.

Milestone

Finish implementing the `get_asteroids` function, write some tests for the related functions in the `test_api.py` file.

hint

- You can import any function from the `api` module by writing `from asteroids.api import something`.

Milestone

Write a short description of your project in the `README.md` file.