Python Packaging Essentials

Joseph Barbier

2025-04-08

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 **10 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 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.

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:

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

```
  { name="author_name", email="email" },
]
dependencies = []
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.urls]
Homepage = "https://github.com/user_name/sunflower"
Issues = "https://github.com/user_name/sunflower/issues"
```

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.

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.

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:

```python
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)
```

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

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

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 Github Actions

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

8 Pre-commit

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

9 API design

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

10 Publish to PyPI

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