

Creating a Python Package

Scientific Programming with Python

Jonas Eschle

University of Zurich
Faculty of Science

16 July 2026



This work is licensed under the *Creative Commons Attribution-ShareAlike 4.0 License*.

Why Package Your Code?

Sound familiar?

- ▶ your functions live in scripts, scattered over project folders

- ▶ to reuse them: copy the file around ... or worse:

```
# From a typical project
```

```
import sys
```

```
sys.path.append("/home/ada/phd/project1/scripts") # only works on my laptop
```

```
sys.path.append("../..../old_project/tools") # only works from this folder
```

- ▶ breaks when you move a folder, change machine or share with a colleague
- ▶ **the fix:** make your code *installable* → create a package

Sound familiar?

- ▶ your functions live in scripts, scattered over project folders
- ▶ to reuse them: copy the file around ... or worse:

```
# From a typical project
```

```
import sys
```

```
sys.path.append("/home/ada/phd/project1/scripts") # only works on my laptop
```

```
sys.path.append("../old_project/tools") # only works from this folder
```

- ▶ breaks when you move a folder, change machine or share with a colleague
- ▶ **the fix:** make your code *installable* → create a package

Sound familiar?

- ▶ your functions live in scripts, scattered over project folders
- ▶ to reuse them: copy the file around ... or worse:

```
# From a typical project
```

```
import sys
```

```
sys.path.append("/home/ada/phd/project1/scripts") # only works on my laptop
```

```
sys.path.append("../old_project/tools") # only works from this folder
```

- ▶ breaks when you move a folder, change machine or share with a colleague
- ▶ **the fix:** make your code *installable* → create a package

Sound familiar?

- ▶ your functions live in scripts, scattered over project folders
- ▶ to reuse them: copy the file around ... or worse:

```
# From a typical project
```

```
import sys
```

```
sys.path.append("/home/ada/phd/project1/scripts") # only works on my laptop
```

```
sys.path.append("../old_project/tools") # only works from this folder
```

- ▶ breaks when you move a folder, change machine or share with a colleague
- ▶ **the fix:** make your code *installable* → create a package

Why create a package?

- ▶ **import from anywhere** – no path hacks, works in every script and notebook
- ▶ **declared dependencies** – installing your package pulls in what it needs
- ▶ **easy to share** – colleagues (and future you) install it with one command
- ▶ **encourages structure** – a natural home for tests, docs and versions

Goal: packaging to *organize* your code.

Publishing to the world has other complications, overview at the end.

Why create a package?

- ▶ **import from anywhere** – no path hacks, works in every script and notebook
- ▶ **declared dependencies** – installing your package pulls in what it needs
- ▶ **easy to share** – colleagues (and future you) install it with one command
- ▶ **encourages structure** – a natural home for tests, docs and versions

Goal: packaging to *organize* your code.

Publishing to the world has other complications, overview at the end.

Why create a package?

- ▶ **import from anywhere** – no path hacks, works in every script and notebook
- ▶ **declared dependencies** – installing your package pulls in what it needs
- ▶ **easy to share** – colleagues (and future you) install it with one command
- ▶ **encourages structure** – a natural home for tests, docs and versions

Goal: packaging to *organize* your code.

Publishing to the world has other complications, overview at the end.

Why create a package?

- ▶ **import from anywhere** – no path hacks, works in every script and notebook
- ▶ **declared dependencies** – installing your package pulls in what it needs
- ▶ **easy to share** – colleagues (and future you) install it with one command
- ▶ **encourages structure** – a natural home for tests, docs and versions

Goal: packaging to *organize* your code.

Publishing to the world has other complications, overview at the end.

Why create a package?

- ▶ **import from anywhere** – no path hacks, works in every script and notebook
- ▶ **declared dependencies** – installing your package pulls in what it needs
- ▶ **easy to share** – colleagues (and future you) install it with one command
- ▶ **encourages structure** – a natural home for tests, docs and versions

Goal: packaging to *organize* your code.

Publishing to the world has other complications, overview at the end.

How far do you need to go?

Packaging is not all-or-nothing – there are three levels of effort.
Every following slide carries its level in the top right corner:

NEED

organize your own code

always worth it – the core of today

SHOULD

use seriously & share

once others (or future you) install it

PRO

publish & maintain

when the world depends on it

Stop at any level – everything below builds on the level above it.

What is a Package?

“Package” means two things

- ▶ **import package** – what you *import*: a directory with an `__init__.py`
- ▶ **distribution package** – what you *install*: an archive containing import packages plus metadata (version, author, dependencies, ...)
- ▶ how they meet: `import` searches the directories on `sys.path`; *installing* places files into one of them, `site-packages`:

```
$ python -c "import numpy; print(numpy.__file__)"  
~/venv/lib/python3.12/site-packages/numpy/__init__.py
```
- ▶ in packaging context, “package” almost always means *distribution package*

“Package” means two things

- ▶ **import package** – what you *import*: a directory with an `__init__.py`
- ▶ **distribution package** – what you *install*: an archive containing import packages plus metadata (version, author, dependencies, ...)
- ▶ how they meet: `import` searches the directories on `sys.path`; *installing* places files into one of them, `site-packages`:

```
$ python -c "import numpy; print(numpy.__file__)"  
~/venv/lib/python3.12/site-packages/numpy/__init__.py
```
- ▶ in packaging context, “package” almost always means *distribution package*

“Package” means two things

- ▶ **import package** – what you *import*: a directory with an `__init__.py`
- ▶ **distribution package** – what you *install*: an archive containing import packages plus metadata (version, author, dependencies, ...)
- ▶ how they meet: `import` searches the directories on `sys.path`; *installing* places files into one of them, `site-packages`:

```
$ python -c "import numpy; print(numpy.__file__)"  
~/venv/lib/python3.12/site-packages/numpy/__init__.py
```
- ▶ in packaging context, “package” almost always means *distribution package*

Distribution packages: wheel and sdist

- ▶ **wheel** (.whl) – a ZIP archive of the files to install; installing = unpacking it into `site-packages`, nothing more
- ▶ **sdist** (.tar.gz) – raw source; a wheel must first be *built* from it (the build backend runs)

```
$ pip download numpy
Saved ./numpy-2.5.1-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl
$ unzip -l numpy-2.5.1-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl
  numpy/__init__.py
  numpy/random/_generator.cpython-312-x86_64-linux-gnu.so
  numpy-2.5.1.dist-info/METADATA
  ...
  (1046 files)
```

- ▶ the name encodes **where the wheel runs**: `cp312` = CPython 3.12, `manylinux_2_28_x86_64` = the platform
- ▶ pure-Python wheels instead say `py3-none-any`: one wheel, runs everywhere

Distribution packages: wheel and sdist

- ▶ **wheel** (.whl) – a ZIP archive of the files to install; installing = unpacking it into `site-packages`, nothing more
- ▶ **sdist** (.tar.gz) – raw source; a wheel must first be *built* from it (the build backend runs)

```
$ pip download numpy
Saved ./numpy-2.5.1-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl
$ unzip -l numpy-2.5.1-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl
  numpy/__init__.py
  numpy/random/_generator.cpython-312-x86_64-linux-gnu.so
  numpy-2.5.1.dist-info/METADATA
  ...
                        (1046 files)
```

- ▶ the name encodes **where the wheel runs**: `cp312` = CPython 3.12, `manylinux_2_28_x86_64` = the platform
- ▶ pure-Python wheels instead say `py3-none-any`: one wheel, runs everywhere

Who does what

- ▶ **build backend** – turns your source tree into sdist + wheel (hatchling, setuptools, flit, uv_build, ...)
- ▶ **build frontend** – calls the backend for you (`python -m build`, `pip`, `uv`)
- ▶ **installer** – downloads wheels and copies them into your environment (`pip`, `uv`)

Standards, not tools

You describe your package *once*, declaratively, in `pyproject.toml` (PEP 517/621).
Any standard-compliant tool can then build and install it – pick whichever you like.

Who does what

- ▶ **build backend** – turns your source tree into sdist + wheel (hatchling, setuptools, flit, uv_build, ...)
- ▶ **build frontend** – calls the backend for you (`python -m build`, `pip`, `uv`)
- ▶ **installer** – downloads wheels and copies them into your environment (`pip`, `uv`)

Standards, not tools

You describe your package *once*, declaratively, in `pyproject.toml` (PEP 517/621).
Any standard-compliant tool can then build and install it – pick whichever you like.

Who does what

- ▶ **build backend** – turns your source tree into sdist + wheel (hatchling, setuptools, flit, uv_build, ...)
- ▶ **build frontend** – calls the backend for you (`python -m build`, `pip`, `uv`)
- ▶ **installer** – downloads wheels and copies them into your environment (`pip`, `uv`)

Standards, not tools

You describe your package *once*, declaratively, in `pyproject.toml` (PEP 517/621).
Any standard-compliant tool can then build and install it – pick whichever you like.

Who does what

- ▶ **build backend** – turns your source tree into sdist + wheel (hatchling, setuptools, flit, uv_build, ...)
- ▶ **build frontend** – calls the backend for you (`python -m build`, `pip`, `uv`)
- ▶ **installer** – downloads wheels and copies them into your environment (`pip`, `uv`)

Standards, not tools

You describe your package *once*, declaratively, in `pyproject.toml` (PEP 517/621). Any standard-compliant tool can then build and install it – pick whichever you like.

A Short History

How we got here

- ▶ **2000** – `distutils` in the standard library: configuration is an executable script, `setup.py`
- ▶ **2003** – PyPI, the Python Package Index, goes online
- ▶ **2004** – `setuptools`: dependencies, `easy_install`, the *egg* format
- ▶ **2008/2012** – `pip` replaces `easy_install`; the *wheel* replaces the *egg*
- ▶ **2015–2021** – the standards era: `pyproject.toml`, pluggable build backends (PEP 517/518), declarative metadata (PEP 621)
- ▶ **today** – one config file, many interoperable tools (`pip`, `build`, `hatch`, `uv`, ...)

How we got here

- ▶ **2000** – `distutils` in the standard library: configuration is an executable script, `setup.py`
- ▶ **2003** – PyPI, the Python Package Index, goes online
- ▶ **2004** – `setuptools`: dependencies, `easy_install`, the *egg* format
- ▶ **2008/2012** – `pip` replaces `easy_install`; the *wheel* replaces the *egg*
- ▶ **2015–2021** – the standards era: `pyproject.toml`, pluggable build backends (PEP 517/518), declarative metadata (PEP 621)
- ▶ **today** – one config file, many interoperable tools (`pip`, `build`, `hatch`, `uv`, ...)

How we got here

- ▶ **2000** – `distutils` in the standard library: configuration is an executable script, `setup.py`
- ▶ **2003** – PyPI, the Python Package Index, goes online
- ▶ **2004** – `setuptools`: dependencies, `easy_install`, the *egg* format
- ▶ **2008/2012** – `pip` replaces `easy_install`; the *wheel* replaces the *egg*
- ▶ **2015–2021** – the standards era: `pyproject.toml`, pluggable build backends (PEP 517/518), declarative metadata (PEP 621)
- ▶ **today** – one config file, many interoperable tools (`pip`, `build`, `hatch`, `uv`, ...)

How we got here

- ▶ **2000** – `distutils` in the standard library: configuration is an executable script, `setup.py`
- ▶ **2003** – PyPI, the Python Package Index, goes online
- ▶ **2004** – `setuptools`: dependencies, `easy_install`, the *egg* format
- ▶ **2008/2012** – `pip` replaces `easy_install`; the *wheel* replaces the egg
- ▶ **2015–2021** – the standards era: `pyproject.toml`, pluggable build backends (PEP 517/518), declarative metadata (PEP 621)
- ▶ **today** – one config file, many interoperable tools (`pip`, `build`, `hatch`, `uv`, ...)

How we got here

- ▶ **2000** – `distutils` in the standard library: configuration is an executable script, `setup.py`
- ▶ **2003** – PyPI, the Python Package Index, goes online
- ▶ **2004** – `setuptools`: dependencies, `easy_install`, the *egg* format
- ▶ **2008/2012** – `pip` replaces `easy_install`; the *wheel* replaces the egg
- ▶ **2015–2021** – the standards era: `pyproject.toml`, pluggable build backends (PEP 517/518), declarative metadata (PEP 621)
- ▶ **today** – one config file, many interoperable tools (`pip`, `build`, `hatch`, `uv`, ...)

How we got here

- ▶ **2000** – `distutils` in the standard library: configuration is an executable script, `setup.py`
- ▶ **2003** – PyPI, the Python Package Index, goes online
- ▶ **2004** – `setuptools`: dependencies, `easy_install`, the *egg* format
- ▶ **2008/2012** – `pip` replaces `easy_install`; the *wheel* replaces the egg
- ▶ **2015–2021** – the standards era: `pyproject.toml`, pluggable build backends (PEP 517/518), declarative metadata (PEP 621)
- ▶ **today** – one config file, many interoperable tools (`pip`, `build`, `hatch`, `uv`, ...)

You will still meet `setup.py`

- ▶ plenty of (scientific) projects still configure themselves the historical way:

```
# setup.py -- the historical way, you will still meet it in older projects
from setuptools import setup, find_packages
```

```
setup(
    name="unithelper",
    version="0.1.0",
    packages=find_packages(),
    install_requires=["numpy"],
)
```

- ▶ reading it: `install_requires` $\approx$ dependencies, `find_packages()` $\approx$ layout
- ▶ **never run it:** `python setup.py install` is gone (removed in `setuptools` $\geq$ 80)

`pip install .` and `pip install -e .` still work fine on such projects – `pip` calls `setuptools` behind the scenes.

You will still meet `setup.py`

- ▶ plenty of (scientific) projects still configure themselves the historical way:

```
# setup.py -- the historical way, you will still meet it in older projects
from setuptools import setup, find_packages
```

```
setup(
    name="unithelper",
    version="0.1.0",
    packages=find_packages(),
    install_requires=["numpy"],
)
```

- ▶ reading it: `install_requires` $\approx$ dependencies, `find_packages()` $\approx$ layout
- ▶ never run it: `python setup.py install` is gone (removed in `setuptools` $\geq$ 80)

`pip install .` and `pip install -e .` still work fine on such projects – `pip` calls `setuptools` behind the scenes.

You will still meet `setup.py`

- ▶ plenty of (scientific) projects still configure themselves the historical way:

```
# setup.py -- the historical way, you will still meet it in older projects
from setuptools import setup, find_packages
```

```
setup(
    name="unithelper",
    version="0.1.0",
    packages=find_packages(),
    install_requires=["numpy"],
)
```

- ▶ reading it: `install_requires` $\approx$ dependencies, `find_packages()` $\approx$ layout
- ▶ **never run it:** `python setup.py install` is gone (removed in `setuptools` $\geq$ 80)

`pip install .` and `pip install -e .` still work fine on such projects – `pip` calls `setuptools` behind the scenes.

You will still meet `setup.py`

- ▶ plenty of (scientific) projects still configure themselves the historical way:

```
# setup.py -- the historical way, you will still meet it in older projects
from setuptools import setup, find_packages
```

```
setup(
    name="unithelper",
    version="0.1.0",
    packages=find_packages(),
    install_requires=["numpy"],
)
```

- ▶ reading it: `install_requires` $\approx$ dependencies, `find_packages()` $\approx$ layout
- ▶ **never run it:** `python setup.py install` is gone (removed in `setuptools` $\geq$ 80)

`pip install .` and `pip install -e .` still work fine on such projects – `pip` calls `setuptools` behind the scenes.

A Minimal Package

The goal

NEED

- ▶ running example: `unithelper`, a tiny library for units and measurements
- ▶ we use the **src layout** – the import package lives under `src/`

```
unithelper
|-- pyproject.toml
`-- src
    |-- unithelper
    |-- core.py
    `-- __init__.py
```

- ▶ your modules, unchanged – plus `__init__.py`
- ▶ one new file: `pyproject.toml`
- ▶ why `src/`? Python puts the *current directory* first on `sys.path` – a top-level `unithelper/` would shadow the installed one; under `src/` you always import what is *installed*

The goal

NEED

- ▶ running example: `unithelper`, a tiny library for units and measurements
- ▶ we use the **src layout** – the import package lives under `src/`

```
unithelper
|-- pyproject.toml
`-- src
    |-- unithelper
    |-- core.py
    `-- __init__.py
```

- ▶ your modules, unchanged – plus `__init__.py`
- ▶ one new file: `pyproject.toml`
- ▶ why `src/`? Python puts the *current directory* first on `sys.path` – a top-level `unithelper/` would shadow the installed one; under `src/` you always import what is *installed*

pyproject.toml – the bare minimum

NEED

- ▶ this file is **all** that is technically required:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

```
[project]
name = "unithelper"
version = "0.1.0"
```

- ▶ [build-system] – which tool turns your source tree into something installable; pip fetches it *itself*, into a temporary isolated environment, whenever it needs to (PEP 517)
- ▶ [project] – metadata; only name and version are mandatory
- ▶ the file is TOML (part 1: `tomllib`) – purely declarative, no code runs to read it

pyproject.toml – the bare minimum

NEED

- ▶ this file is **all** that is technically required:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

```
[project]
name = "unithelper"
version = "0.1.0"
```

- ▶ [build-system] – which tool turns your source tree into something installable; pip fetches it *itself*, into a temporary isolated environment, whenever it needs to (PEP 517)
- ▶ [project] – metadata; only name and version are mandatory
- ▶ the file is TOML (part 1: toml1lib) – purely declarative, no code runs to read it

Install it – editable

NEED

- ▶ install it – directly from the source folder:

```
$ pip install -e .  
Obtaining file:///~/unithelper  
Building editable for unithelper (pyproject.toml) ... done  
Successfully installed unithelper-0.1.0
```

- ▶ `-e` = **editable**: what lands in `site-packages` is *not a copy* of your code – it is a link back to `src/`:

```
$ ls ~/.venv/lib/python3.12/site-packages/ | grep unithelper  
_editable_impl_unithelper.pth  
unithelper-0.1.0.dist-info  
$ cat ~/.venv/lib/python3.12/site-packages/_editable_impl_unithelper.pth  
~/unithelper/src
```

- ▶ Python reads `.pth` files at startup → every `import unithelper` finds your working copy: edits are active immediately, no re-install
- ▶ without `-e`: files are *copied* – a snapshot; right for *users*, stale after your next edit

Install it – editable

NEED

- ▶ install it – directly from the source folder:

```
$ pip install -e .  
Obtaining file:///~/unithelper  
Building editable for unithelper (pyproject.toml) ... done  
Successfully installed unithelper-0.1.0
```

- ▶ `-e` = **editable**: what lands in `site-packages` is *not a copy* of your code – it is a link back to `src/`:

```
$ ls ~/.venv/lib/python3.12/site-packages/ | grep unithelper  
_editable_impl_unithelper.pth  
unithelper-0.1.0.dist-info  
$ cat ~/.venv/lib/python3.12/site-packages/_editable_impl_unithelper.pth  
~/unithelper/src
```

- ▶ Python reads `.pth` files at startup → every `import unithelper` finds your working copy: edits are active immediately, no re-install
- ▶ without `-e`: files are *copied* – a snapshot; right for *users*, stale after your next edit

It works – and you could stop here

NEED

```
$ cd ~/somewhere/else
$ python
>>> import unithelper
>>> unithelper.celsius_to_kelvin(21.5)
294.65
```

```
$ pip show unithelper
Name: unithelper
Version: 0.1.0
Summary:
Location: ~/.venv/lib/python3.12/site-packages
Editable project location: ~/unithelper
Requires:
```

- ▶ no path hacks: everything running in this environment – scripts, notebooks, consoles – finds unithelper via site-packages
- ▶ pip manages it like any other package; the empty fields are metadata we did not declare – yet

Checkpoint

Layout + 7 lines of TOML + `pip install -e .` organizes your own code.
Everything that follows is about *publishing* it.

It works – and you could stop here

NEED

```
$ cd ~/somewhere/else
$ python
>>> import unithelper
>>> unithelper.celsius_to_kelvin(21.5)
294.65
```

```
$ pip show unithelper
Name: unithelper
Version: 0.1.0
Summary:
Location: ~/.venv/lib/python3.12/site-packages
Editable project location: ~/unithelper
Requires:
```

- ▶ no path hacks: everything running in this environment – scripts, notebooks, consoles – finds unithelper via site-packages
- ▶ pip manages it like any other package; the empty fields are metadata we did not declare – yet

Checkpoint

Layout + 7 lines of TOML + `pip install -e .` organizes your own code.
Everything that follows is about *publishing* it.

Metadata & Dependencies

The grown-up package

SHOULD

```
unithelper
|-- LICENSE
|-- pyproject.toml
|-- README.md
|-- src
|   |-- unithelper
|   |   |-- cli.py
|   |   |-- core.py
|   |   |-- __init__.py
|   |   |-- plotting.py
|   |   |-- stats.py
|-- tests
|   |-- test_core.py
```

- ▶ `README.md` – what/why/how; shown by GitHub and PyPI
- ▶ `LICENSE` – without one, nobody may legally reuse your code
- ▶ `tests/` – pytest tests, run against the *installed* package
- ▶ several modules – users see one import package

The grown-up package

SHOULD

```
unithelper
|-- LICENSE
|-- pyproject.toml
|-- README.md
|-- src
|   |-- unithelper
|   |   |-- cli.py
|   |   |-- core.py
|   |   |-- __init__.py
|   |   |-- plotting.py
|   |   |-- stats.py
|-- tests
|   |-- test_core.py
```

- ▶ `README.md` – what/why/how; shown by GitHub and PyPI
- ▶ `LICENSE` – without one, nobody may legally reuse your code
- ▶ `tests/` – pytest tests, run against the *installed* package
- ▶ several modules – users see one import package

The grown-up package

SHOULD

```
unithelper
|-- LICENSE
|-- pyproject.toml
|-- README.md
|-- src
|   |-- unithelper
|   |   |-- cli.py
|   |   |-- core.py
|   |   |-- __init__.py
|   |   |-- plotting.py
|   |   |-- stats.py
|-- tests
|   |-- test_core.py
```

- ▶ `README.md` – what/why/how; shown by GitHub and PyPI
- ▶ `LICENSE` – without one, nobody may legally reuse your code
- ▶ `tests/` – pytest tests, run against the *installed* package
- ▶ several modules – users see one import package

The grown-up package

SHOULD

```
unithelper
|-- LICENSE
|-- pyproject.toml
|-- README.md
|-- src
|   |-- unithelper
|   |   |-- cli.py
|   |   |-- core.py
|   |   |-- __init__.py
|   |   |-- plotting.py
|   |   |-- stats.py
|-- tests
|   |-- test_core.py
```

- ▶ `README.md` – what/why/how; shown by GitHub and PyPI
- ▶ `LICENSE` – without one, nobody may legally reuse your code
- ▶ `tests/` – pytest tests, run against the *installed* package
- ▶ several modules – users see one import package

Describe your package

SHOULD

- ▶ the [project] table grows with real metadata:

```
[project]
name = "unithelper"
version = "0.1.0"
description = "Tiny helpers for units and measurements"
readme = "README.md"
requires-python = ">=3.11"
license = "MIT"
license-files = ["LICENSE"]
authors = [{ name = "Ada Lovelace", email = "ada.lovelace@uzh.ch" }]
```

- ▶ `requires-python`: the *oldest* Python you support – lower bound only, **no upper cap**
- ▶ `license`: an SPDX expression like "MIT" plus the license file (the modern standard, PEP 639)

Describe your package

SHOULD

- ▶ the `[project]` table grows with real metadata:

```
[project]
name = "unithelper"
version = "0.1.0"
description = "Tiny helpers for units and measurements"
readme = "README.md"
requires-python = ">=3.11"
license = "MIT"
license-files = ["LICENSE"]
authors = [{ name = "Ada Lovelace", email = "ada.lovelace@uzh.ch" }]
```

- ▶ `requires-python`: the *oldest* Python you support – lower bound only, **no upper cap**
- ▶ `license`: an SPDX expression like "MIT" plus the license file (the modern standard, PEP 639)

Describe your package

SHOULD

- ▶ the [project] table grows with real metadata:

```
[project]
name = "unithelper"
version = "0.1.0"
description = "Tiny helpers for units and measurements"
readme = "README.md"
requires-python = ">=3.11"
license = "MIT"
license-files = ["LICENSE"]
authors = [{ name = "Ada Lovelace", email = "ada.lovelace@uzh.ch" }]
```

- ▶ requires-python: the *oldest* Python you support – lower bound only, **no upper cap**
- ▶ license: an SPDX expression like "MIT" plus the license file (the modern standard, PEP 639)

Declare your dependencies

SHOULD

- ▶ what must be installed so your package works?

```
dependencies = [  
    "numpy>=1.26",  
]
```

- ▶ this list travels *inside* the package metadata – wherever it is installed, the installer first resolves and installs the dependencies, recursively (numpy's dependencies too)
- ▶ version specifiers: `>=`, `<`, `==`, `!=`, `~=`
- ▶ `numpy>=1.26` reads: “the oldest numpy I actually support”
– ideally the oldest version you tested against

Declare your dependencies

SHOULD

- ▶ what must be installed so your package works?

```
dependencies = [  
    "numpy>=1.26",  
]
```

- ▶ this list travels *inside* the package metadata – wherever it is installed, the installer first resolves and installs the dependencies, recursively (numpy's dependencies too)
- ▶ version specifiers: `>=`, `<`, `==`, `!=`, `~=`
- ▶ `numpy>=1.26` reads: “the oldest numpy I actually support”
– ideally the oldest version you tested against

Declare your dependencies

SHOULD

- ▶ what must be installed so your package works?

```
dependencies = [  
    "numpy>=1.26",  
]
```

- ▶ this list travels *inside* the package metadata – wherever it is installed, the installer first resolves and installs the dependencies, recursively (numpy's dependencies too)
- ▶ version specifiers: >=, <, ==, !=, ~=
- ▶ `numpy>=1.26` reads: “the oldest numpy I actually support”
– ideally the oldest version you tested against

Optional and development dependencies

SHOULD

- ▶ **extras** – optional *features* your users opt into:

```
[project.optional-dependencies]  
plot = ["matplotlib>=3.8"]
```

- ▶ **dependency groups** – for *developers* only, never shipped to users:

```
[dependency-groups]  
test = ["pytest>=8"]
```

```
$ pip install -e ".[plot]"  
Collecting matplotlib>=3.8 (from unithelper==0.1.0)  
Successfully installed ... matplotlib-3.11.0 ...  
$ pip install --group test  
Collecting pytest>=8  
Successfully installed iniconfig-2.3.0 pluggy-1.6.0 pygments-2.20.0 pytest-9.1.1
```

Optional and development dependencies

SHOULD

- ▶ **extras** – optional *features* your users opt into:

```
[project.optional-dependencies]  
plot = ["matplotlib>=3.8"]
```

- ▶ **dependency groups** – for *developers* only, never shipped to users:

```
[dependency-groups]  
test = ["pytest>=8"]
```

```
$ pip install -e ".[plot]"  
Collecting matplotlib>=3.8 (from unithelper==0.1.0)  
Successfully installed ... matplotlib-3.11.0 ...  
$ pip install --group test  
Collecting pytest>=8  
Successfully installed iniconfig-2.3.0 pluggy-1.6.0 pygments-2.20.0 pytest-9.1.1
```

Optional and development dependencies

SHOULD

- ▶ **extras** – optional *features* your users opt into:

```
[project.optional-dependencies]  
plot = ["matplotlib>=3.8"]
```

- ▶ **dependency groups** – for *developers* only, never shipped to users:

```
[dependency-groups]  
test = ["pytest>=8"]
```

```
$ pip install -e ".[plot]"  
Collecting matplotlib>=3.8 (from unithelper==0.1.0)  
Successfully installed ... matplotlib-3.11.0 ...  
$ pip install --group test  
Collecting pytest>=8  
Successfully installed iniconfig-2.3.0 pluggy-1.6.0 pygments-2.20.0 pytest-9.1.1
```

How tight should version constraints be?

SHOULD

- ▶ **library** (others install it next to who-knows-what):
 - ▶ lower bounds only: `numpy>=1.26`
 - ▶ **no speculative upper caps** (`numpy<3`): you cannot know the future, but caps make environments unsolvable
- ▶ **application / analysis** (lives in its own environment):
 - ▶ reproducibility wins: pin everything via a *lock file* (`uv.lock`, `pylock.toml`)
 - ▶ lock file = exact versions of the full environment, kept next to the code

Rule of thumb

In `pyproject.toml`: lower bounds only – fix incompatibilities when they actually appear, not preemptively.

How tight should version constraints be?

SHOULD

- ▶ **library** (others install it next to who-knows-what):
 - ▶ lower bounds only: `numpy>=1.26`
 - ▶ **no speculative upper caps** (`numpy<3`): you cannot know the future, but caps make environments unsolvable
- ▶ **application / analysis** (lives in its own environment):
 - ▶ reproducibility wins: pin everything via a *lock file* (`uv.lock`, `pylock.toml`)
 - ▶ lock file = exact versions of the full environment, kept next to the code

Rule of thumb

In `pyproject.toml`: lower bounds only – fix incompatibilities when they actually appear, not preemptively.

How tight should version constraints be?

SHOULD

- ▶ **library** (others install it next to who-knows-what):
 - ▶ lower bounds only: `numpy>=1.26`
 - ▶ **no speculative upper caps** (`numpy<3`): you cannot know the future, but caps make environments unsolvable
- ▶ **application / analysis** (lives in its own environment):
 - ▶ reproducibility wins: pin everything via a *lock file* (`uv.lock`, `pylock.toml`)
 - ▶ lock file = exact versions of the full environment, kept next to the code

Rule of thumb

In `pyproject.toml`: lower bounds only – fix incompatibilities when they actually appear, not preemptively.

Command line scripts

SHOULD

- ▶ remember argparse/click from part 1? A package turns them into real commands:

```
[project.scripts]  
convert-temp = "unithelper.cli:main"
```

- ▶ `command = "module.path:function"` – the function is ordinary Python:

```
def main():  
    parser = argparse.ArgumentParser(description="Convert between temperature scales.")  
    parser.add_argument("value", type=float)  
    parser.add_argument("scale_from", choices="CKF", metavar="from")  
    parser.add_argument("scale_to", choices="CKF", metavar="to")
```

- ▶ at install time, pip generates a real `convert-temp` executable in the environment's `bin/` – a tiny wrapper that imports `unithelper.cli` and calls `main()`

```
$ convert-temp 300 K C  
300 K = 26.85 C  
$ convert-temp 25 C F  
25 C = 77 F
```

Command line scripts

SHOULD

- ▶ remember argparse/click from part 1? A package turns them into real commands:

```
[project.scripts]  
convert-temp = "unithelper.cli:main"
```

- ▶ `command = "module.path:function"` – the function is ordinary Python:

```
def main():  
    parser = argparse.ArgumentParser(description="Convert between temperature scales.")  
    parser.add_argument("value", type=float)  
    parser.add_argument("scale_from", choices="CKF", metavar="from")  
    parser.add_argument("scale_to", choices="CKF", metavar="to")
```

- ▶ at install time, pip generates a real `convert-temp` executable in the environment's `bin/` – a tiny wrapper that imports `unithelper.cli` and calls `main()`

```
$ convert-temp 300 K C  
300 K = 26.85 C  
$ convert-temp 25 C F  
25 C = 77 F
```

Command line scripts

SHOULD

- ▶ remember argparse/click from part 1? A package turns them into real commands:

```
[project.scripts]  
convert-temp = "unithelper.cli:main"
```

- ▶ `command = "module.path:function"` – the function is ordinary Python:

```
def main():  
    parser = argparse.ArgumentParser(description="Convert between temperature scales.")  
    parser.add_argument("value", type=float)  
    parser.add_argument("scale_from", choices="CKF", metavar="from")  
    parser.add_argument("scale_to", choices="CKF", metavar="to")
```

- ▶ at install time, pip generates a real `convert-temp` executable in the environment's `bin/` – a tiny wrapper that imports `unithelper.cli` and calls `main()`

```
$ convert-temp 300 K C  
300 K = 26.85 C  
$ convert-temp 25 C F  
25 C = 77 F
```

Building & Sharing

Build it

SHOULD

- ▶ so far we installed from the source folder – to *hand the package to others*, build distribution files:

```
$ python -m build
* Creating isolated environment: venv+pip...
* Building sdist...
* Building wheel...
Successfully built unithelper-0.1.0.tar.gz and unithelper-0.1.0-py3-none-any.whl
$ ls dist/
unithelper-0.1.0-py3-none-any.whl
unithelper-0.1.0.tar.gz
```

- ▶ the frontend (build) reads [build-system] and runs hatchling in a throw-away environment (*Creating isolated environment*) that contains *only* the declared build requirements – the build cannot silently depend on what you happen to have installed
- ▶ py3-none-any: pure Python – this one wheel runs everywhere

Build it

SHOULD

- ▶ so far we installed from the source folder – to *hand the package to others*, build distribution files:

```
$ python -m build
* Creating isolated environment: venv+pip...
* Building sdist...
* Building wheel...
Successfully built unithelper-0.1.0.tar.gz and unithelper-0.1.0-py3-none-any.whl
$ ls dist/
unithelper-0.1.0-py3-none-any.whl
unithelper-0.1.0.tar.gz
```

- ▶ the frontend (build) reads [build-system] and runs hatchling in a throw-away environment (*Creating isolated environment*) that contains *only* the declared build requirements – the build cannot silently depend on what you happen to have installed
- ▶ py3-none-any: pure Python – this one wheel runs everywhere

Build it

SHOULD

- ▶ so far we installed from the source folder – to *hand the package to others*, build distribution files:

```
$ python -m build
* Creating isolated environment: venv+pip...
* Building sdist...
* Building wheel...
Successfully built unithelper-0.1.0.tar.gz and unithelper-0.1.0-py3-none-any.whl
$ ls dist/
unithelper-0.1.0-py3-none-any.whl
unithelper-0.1.0.tar.gz
```

- ▶ the frontend (build) reads [build-system] and runs hatchling in a throw-away environment (*Creating isolated environment*) that contains *only* the declared build requirements – the build cannot silently depend on what you happen to have installed
- ▶ py3-none-any: pure Python – this one wheel runs everywhere

Inside the wheel

SHOULD

```
$ unzip -l dist/unithelper-0.1.0-py3-none-any.whl
unithelper/__init__.py
unithelper/cli.py
unithelper/core.py
unithelper/plotting.py
unithelper/stats.py
unithelper-0.1.0.dist-info/METADATA
unithelper-0.1.0.dist-info/WHEEL
unithelper-0.1.0.dist-info/entry_points.txt
unithelper-0.1.0.dist-info/licenses/LICENSE
unithelper-0.1.0.dist-info/RECORD
```

```
$ unzip -p dist/*.whl "*/METADATA"
Name: unithelper
Version: 0.1.0
Summary: Tiny helpers for units and measurements
License-Expression: MIT
Requires-Python: >=3.11
Requires-Dist: numpy>=1.26
Provides-Extra: plot
Requires-Dist: matplotlib>=3.8; extra == 'plot'
```

- ▶ your import package, exactly as it will land in `site-packages`
- ▶ plus `dist-info`: `METADATA` (name, version, `Requires-Dist`), `entry_points.txt` (convert-temp), `RECORD` (every installed file – how pip uninstalls cleanly)

Inside the wheel

SHOULD

```
$ unzip -l dist/unithelper-0.1.0-py3-none-any.whl
unithelper/__init__.py
unithelper/cli.py
unithelper/core.py
unithelper/plotting.py
unithelper/stats.py
unithelper-0.1.0.dist-info/METADATA
unithelper-0.1.0.dist-info/WHEEL
unithelper-0.1.0.dist-info/entry_points.txt
unithelper-0.1.0.dist-info/licenses/LICENSE
unithelper-0.1.0.dist-info/RECORD
```

```
$ unzip -p dist/*.whl "*/METADATA"
Name: unithelper
Version: 0.1.0
Summary: Tiny helpers for units and measurements
License-Expression: MIT
Requires-Python: >=3.11
Requires-Dist: numpy>=1.26
Provides-Extra: plot
Requires-Dist: matplotlib>=3.8; extra == 'plot'
```

- ▶ your import package, exactly as it will land in site-packages
- ▶ plus dist-info: METADATA (name, version, Requires-Dist), entry_points.txt (convert-temp), RECORD (every installed file – how pip uninstalls cleanly)

Share it

SHOULD

- ▶ send the wheel to a colleague – mail, USB stick, group share, ...:

```
$ pip install unithelper-0.1.0-py3-none-any.whl
Processing ./dist/unithelper-0.1.0-py3-none-any.whl
Collecting numpy>=1.26 (from unithelper==0.1.0)
Successfully installed numpy-2.5.1 unithelper-0.1.0
$ convert-temp 300 K C
300 K = 26.85 C
```

- ▶ dependencies and the command arrive automatically – **metadata at work**
- ▶ pick meaningful versions – **SemVer** MAJOR.MINOR.PATCH:
breaking change / new feature / bug fix

Share it

SHOULD

- ▶ send the wheel to a colleague – mail, USB stick, group share, ...:

```
$ pip install unithelper-0.1.0-py3-none-any.whl
Processing ./dist/unithelper-0.1.0-py3-none-any.whl
Collecting numpy>=1.26 (from unithelper==0.1.0)
Successfully installed numpy-2.5.1 unithelper-0.1.0
$ convert-temp 300 K C
300 K = 26.85 C
```

- ▶ dependencies and the command arrive automatically – **metadata at work**
- ▶ pick meaningful versions – **SemVer** MAJOR.MINOR.PATCH:
breaking change / new feature / bug fix

The one-stop shop: uv

SHOULD

- ▶ one fast tool for environments, dependencies, building and publishing:

```
$ uv init --package mypkg && cd mypkg
Initialized project `mypkg` at `~/mypkg`
$ uv add numpy
Creating virtual environment at: .venv
Installed 2 packages
+ mypkg==0.1.0 (from file://~/mypkg)
+ numpy==2.5.1
$ uv build
Successfully built dist/mypkg-0.1.0.tar.gz
Successfully built dist/mypkg-0.1.0-py3-none-any.whl
```

- ▶ `uv add` updates `pyproject.toml` *and* installs, `uv sync` recreates the environment, `uv lock` pins it
- ▶ same standards underneath: `pyproject.toml`, wheels, sdists – everything you just learned applies unchanged

The one-stop shop: uv

SHOULD

- ▶ one fast tool for environments, dependencies, building and publishing:

```
$ uv init --package mypkg && cd mypkg
Initialized project `mypkg` at `~/mypkg`
$ uv add numpy
Creating virtual environment at: .venv
Installed 2 packages
+ mypkg==0.1.0 (from file://~/mypkg)
+ numpy==2.5.1
$ uv build
Successfully built dist/mypkg-0.1.0.tar.gz
Successfully built dist/mypkg-0.1.0-py3-none-any.whl
```

- ▶ `uv add` updates `pyproject.toml` *and* installs, `uv sync` recreates the environment, `uv lock` pins it
- ▶ same standards underneath: `pyproject.toml`, wheels, sdists – everything you just learned applies unchanged

How Real Packages Do It

Publishing to PyPI

PRO

- ▶ PyPI is “just” an index of wheels and sdist – `pip install` searches it, downloads, installs
- ▶ publishing yourself:
 - ▶ create an account, choose a *unique* name
 - ▶ upload your dist/ files: `twine upload dist/*` (or `uv publish`)
 - ▶ practice on test.pypi.org first
- ▶ once published: **a version is immutable** – you can only add new ones

Publish when others should `pip install` your package without asking you – not before it has a license, a README and tests.

Publishing to PyPI

PRO

- ▶ PyPI is “just” an index of wheels and sdist – `pip install` searches it, downloads, installs
- ▶ publishing yourself:
 - ▶ create an account, choose a *unique* name
 - ▶ upload your dist/ files: `twine upload dist/*` (or `uv publish`)
 - ▶ practice on test.pypi.org first
- ▶ once published: **a version is immutable** – you can only add new ones

Publish when others should `pip install` your package without asking you – not before it has a license, a README and tests.

Publishing to PyPI

PRO

- ▶ PyPI is “just” an index of wheels and sdist – `pip install` searches it, downloads, installs
- ▶ publishing yourself:
 - ▶ create an account, choose a *unique* name
 - ▶ upload your dist/ files: `twine upload dist/*` (or `uv publish`)
 - ▶ practice on test.pypi.org first
- ▶ once published: **a version is immutable** – you can only add new ones

Publish when others should `pip install` your package without asking you – not before it has a license, a README and tests.

How real packages publish: from CI

PRO

- ▶ releases are built and uploaded by *CI*, not from a laptop:

```
# .github/workflows/publish.yml
name: publish
on:
  release:
    types: [published]

jobs:
  publish:
    runs-on: ubuntu-latest
    environment: pypi
    permissions:
      id-token: write      # <- "trusted publishing", no password/token needed
    steps:
      - uses: actions/checkout@v4
      - run: pipx run build
      - uses: pypa/gh-action-pypi-publish@release/v1
```

- ▶ **trusted publishing**: PyPI trusts your GitHub repository directly – no passwords or tokens to leak

How real packages test

PRO

- ▶ every push is tested on all supported platforms and Python versions:

```
# .github/workflows/tests.yml
name: tests
on: [push, pull_request]

jobs:
  test:
    runs-on: ${ matrix.os }
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        python-version: ["3.11", "3.12", "3.13"]
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${ matrix.python-version }
      - run: pip install -e . --group test
      - run: pytest
```

- ▶ this matrix is what requires-python = ">=3.11" *promises* – CI is where the promise is kept

When your code is not pure Python

PRO

- ▶ compiled extensions (C, C++, Rust, Cython) need compiling – *someone* has to do it:
 - ▶ **the user**, from the sdist: needs compilers and libraries installed – often fails
 - ▶ **you, in advance**: one wheel per platform × Python version – remember numpy's `cp312-manylinux_x86_64`
- ▶ `cibuildwheel` builds the whole matrix on CI – this is how numpy ships ~40 wheels per release
- ▶ a different build backend does the compiling (`scikit-build-core`, `meson-python`, `maturin`) – `pyproject.toml` stays the same

When your code is not pure Python

PRO

- ▶ compiled extensions (C, C++, Rust, Cython) need compiling – *someone* has to do it:
 - ▶ **the user**, from the sdist: needs compilers and libraries installed – often fails
 - ▶ **you, in advance**: one wheel per platform × Python version – remember numpy's `cp312-manylinux_x86_64`
- ▶ `cibuildwheel` builds the whole matrix on CI – this is how numpy ships ~40 wheels per release
- ▶ a different build backend does the compiling (`scikit-build-core`, `meson-python`, `maturin`) – `pyproject.toml` stays the same

When your code is not pure Python

PRO

- ▶ compiled extensions (C, C++, Rust, Cython) need compiling – *someone* has to do it:
 - ▶ **the user**, from the sdist: needs compilers and libraries installed – often fails
 - ▶ **you, in advance**: one wheel per platform × Python version – remember numpy's `cp312-manylinux_x86_64`
- ▶ `cibuildwheel` builds the whole matrix on CI – this is how numpy ships ~40 wheels per release
- ▶ a different build backend does the compiling (`scikit-build-core`, `meson-python`, `maturin`) – `pyproject.toml` stays the same

Beyond PyPI: conda

PRO

- ▶ wheels can only ship what fits in a Python package – not compilers, CUDA, ROOT, ...
- ▶ **conda** packages *anything*, Python is just another package; **conda-forge**: community-maintained recipes, builds on shared CI
- ▶ typical scientific setup: conda/mamba/pixi environment for the heavy binaries, pip/uv for the pure-Python rest
- ▶ publish to conda-forge *in addition to* PyPI when your users live there

Beyond PyPI: conda

PRO

- ▶ wheels can only ship what fits in a Python package – not compilers, CUDA, ROOT, ...
- ▶ **conda** packages *anything*, Python is just another package; **conda-forge**: community-maintained recipes, builds on shared CI
- ▶ typical scientific setup: conda/mamba/pixi environment for the heavy binaries, pip/uv for the pure-Python rest
- ▶ publish to conda-forge *in addition to* PyPI when your users live there

Beyond PyPI: conda

PRO

- ▶ wheels can only ship what fits in a Python package – not compilers, CUDA, ROOT, ...
- ▶ **conda** packages *anything*, Python is just another package; **conda-forge**: community-maintained recipes, builds on shared CI
- ▶ typical scientific setup: conda/mamba/pixi environment for the heavy binaries, pip/uv for the pure-Python rest
- ▶ publish to conda-forge *in addition to* PyPI when your users live there

Beyond PyPI: conda

PRO

- ▶ wheels can only ship what fits in a Python package – not compilers, CUDA, ROOT, ...
- ▶ **conda** packages *anything*, Python is just another package; **conda-forge**: community-maintained recipes, builds on shared CI
- ▶ typical scientific setup: conda/mamba/pixi environment for the heavy binaries, pip/uv for the pure-Python rest
- ▶ publish to conda-forge *in addition to* PyPI when your users live there

Where to learn more

PRO

- ▶ the official tutorial: <https://packaging.python.org/en/latest/tutorials/packaging-projects/>
- ▶ packaging for scientists, highly recommended:
<https://learn.scientific-python.org/development/>
- ▶ start new packages from a template: <https://github.com/scientific-python/cookie>
- ▶ why no upper bounds – the definitive rant:
<https://iscinumpy.dev/post/bound-version-constraints/>