Lecture 12 — From scripts to an installable package

C1 Research Computing — MPhil in Data Intensive Science, Michaelmas 2026

Contents

1 The import system **2**
- 1.1 The import search path . . . 2
- 1.2 Packages and relative imports . . . 3
- 1.3 The src layout . . . 5

2 Project metadata and installation **5**
- 2.1 Distributions and the project file . . . 6
- 2.2 Version specifiers and optional dependencies . . . 7
- 2.3 Editable installs and lock files . . . 8

3 Console commands and the public interface **9**
- 3.1 Console commands . . . 10
- 3.2 The public interface . . . 11
- 3.3 Semantic versions . . . 12

4 Wheels **13**
- 4.1 The contents of a wheel . . . 13
- 4.2 Package data . . . 14
- 4.3 Compatibility tags . . . 15

Troubleshooting **17**

Key points **18**

Exercises **19**

Solutions to the exercises **21**

References **24**

A program written as a few scripts runs only from the folder that holds the scripts. A Python package is a collection of modules, and possibly of other packages, organised under one name. Code imports the modules by that name. A package can be installed into an environment. Python then imports the installed package from any folder.

Section 1 shows how `import` finds a module. Section 1 also explains why a project keeps its package in a folder `src/`.

Section 2 treats the project file `pyproject.toml`, the version specifiers of requirements and the editable install. Section 3 treats console commands, the public interface of a package and semantic versions. Section 4 opens a wheel, the archive that pip installs, and reads the tags in its name.

The examples come from the standard library, python-dotenv, NumPy, SciPy and Matplotlib. The course project is packaged in the same steps, at the tags `lecture-12-start` and `lecture-12`. The sessions typed at the prompt are stored in the files of `sessions/lecture12/`.

The command `python -m doctest` replays the session files. The outputs were produced with Python 3.13 and pip 25.3.

1 The import system

A program split into several files imports its other files each time the program runs. The folders that Python searches decide two things. The first is whether the program runs from one folder only. The second is whether the program uses the files of the folder it runs from or the files installed in its environment.

1.1 The import search path

Definition 1.1: Import search path

The **import search path** is a list of folders, `sys.path`. The statement `import name` uses `sys.path` in three steps.

1. Python searches the folders of `sys.path` in order.
2. Python imports the first module or package called `name` that Python finds.
3. If no folder holds a module or package called `name`, Python raises `ModuleNotFoundError`.

The list starts with one folder. For `python script.py`, the first folder is the folder of the script. For `python -c`, `python -m` and the prompt, the first folder is the current folder. For example, under `python -c`, `sys.path[0]` is ''. The empty string stands for the current folder.

After the first folder come the folders of the environment variable `PYTHONPATH`. Then comes the standard library. Last comes `site-packages`. pip installs the packages of the active environment into `site-packages`. For example, `"site-packages" in numpy.__file__` is `True`.

Table 1 lists the entries of `sys.path` in the environment of the course project, with the purpose of each entry.

Table 1: The entries of `sys.path` in the environment of the course project, in order, with Python 3.13 on macOS.

entry	purpose
`.../lib/python313.zip`	an optional zip archive of the standard library, often absent
`.../lib/python3.13`	the standard library: `os`, `pathlib`, `json`, ...
`.../lib/python3.13/lib-dynload`	the compiled modules of the standard library, files ending in `.so`
''	the current folder
`.../.venv/lib/python3.13/site-packages`	the packages installed in the environment
`.../attractor/src`	the folder `src/` of the project, added by the editable install (Section 2.3)

In Listing 1, a new folder holds a file `random.py`. The command `python -c "import random"` is run in the new folder. Python then imports the file `random.py` of the new folder instead of the module `random` of the standard library.

The same command run in another folder imports the module `random` of the standard library. Importing the standard module `random` prints nothing.

The function `importlib.util.find_spec` returns `None` for a name that no folder holds. Python raises `ModuleNotFoundError` at the import of a name that no folder holds (`sessions/lecture12/imports.txt`, lines 14–19, at `lecture-12`).

```
>>> import pathlib, subprocess, sys, tempfile
>>> folder = pathlib.Path(tempfile.mkdtemp())
>>> _ = (folder / "random.py").write_text('print("random.py of this folder")')
>>> run = [sys.executable, "-c", "import random"]
>>> subprocess.run(run, cwd=folder, capture_output=True, text=True).stdout
'random.py of this folder\n'
>>> subprocess.run(run, capture_output=True, text=True).stdout
''
```

Listing 1: A file of the current folder found before the standard library: `sessions/lecture12/shadow.txt`, lines 4–11, at `lecture-12`.

A script that imports another file of its project depends on the order of `sys.path`. Take the command `python scripts/run.py`. The folder `scripts/` is then first on `sys.path`. So the script finds only the modules of the folder `scripts/`.

In data analysis, a notebook may run in a directory containing a local copy of a module, such as `simulate.py`. If that directory appears before `site-packages` in `sys.path`, Python imports the local copy instead of the installed version. The analysis then uses the local code, which may differ from the intended version, without raising an error.

For example, in the root of the course project, `import simulate` runs the script `simulate.py` and binds the name `simulate` only. The name `H` stays unbound. After `from simulate import H`, `H` is `0.05`.

> A script named after a module, such as `random.py` or `numpy.py`, shadows that module for every program started in its folder. Python then raises the error later, at the use of a missing attribute. For example, in the folder of Listing 1, `random.randint(1, 6)` raises `AttributeError: module 'random' has no attribute 'randint'`. Python 3.13 adds a hint to rename the file.

1.2 Packages and relative imports

Definition 1.2: Package

A **package** is a module that can contain other modules, its **submodules**, and other packages, its **subpackages**. A regular package is a folder that holds a file `__init__.py`. Importing the package runs the file `__init__.py`.

The package stores its search locations in the attribute `__path__`, usually a list holding the folder of the package. Python searches for the submodules of the package in these locations. The submodule `mod` of `pkg` has the full name `pkg.mod`. Importing `pkg.mod` runs `pkg/__init__.py` first.

An **absolute import** names a module by its full name, as in `from pkg.mod import f`. A **relative import** names a module from the package of the importing module:

- one leading dot refers to the package of the importing module, as in `from .mod import f`;
- two dots refer to the parent package, as in `from ..mod import f`;
- three dots refer to the parent of the parent.

For example, `importlib.util.resolve_name("..mod", "pkg.sub")` is `'pkg.mod'`.

The module `json` of the standard library is a package. The module `math` is a single module,

without `__path__` (Listing 2).

`json.decoder` is a submodule. The file `json/__init__.py` imports `json.decoder` by the relative name `.decoder`. The function `importlib.util.resolve_name` converts the relative name `.decoder` into the full name `json.decoder`.

```
>>> import importlib.util, json, math
>>> hasattr(json, "__path__"), hasattr(math, "__path__")
(True, False)
>>> json.__file__.endswith("json/__init__.py")
True
>>> import json.decoder
>>> json.decoder.__name__, json.decoder.__package__
('json.decoder', 'json')
>>> importlib.util.resolve_name(".decoder", "json")
'json.decoder'
```

Listing 2: A package, a module and a submodule of the standard library: `sessions/lecture12/imports.txt`, lines 4–13, at `lecture-12`.

Listing 3 shows the three levels of relative imports in a package of three levels. The module `linear.py` has the full name `pkg.analysis.models.linear`. For example, resolving `"..fitting"` from the package `pkg.analysis.models` gives `'pkg.analysis.fitting'`.

```
pkg/
├── __init__.py
├── utils.py
└── analysis/
    ├── __init__.py
    ├── fitting.py
    └── models/
        ├── __init__.py
        ├── other.py
        └── linear.py

# in pkg/analysis/models/linear.py
from . import other       # pkg.analysis.models.other
from .. import fitting    # pkg.analysis.fitting
from ... import utils     # pkg.utils
```

Listing 3: An example package of three levels, and the relative imports of its module `pkg/analysis/models/linear.py`.

A module of a package run as a file, as in `python src/attractor/cli.py`, belongs to no package. At the first relative import of the module, Python raises `ImportError: attempted relative import with no known parent package`. The command `python -m attractor.cli` runs the module `cli` inside the package `attractor`.

In `attractor`

At `lecture-12` the modules of the course project lie in the package folder `src/attractor/`, with an `__init__.py`. The modules import each other by relative imports, such as `from .trajectory import Trajectory` in `integrators.py` (`src/attractor/integrators.py`, lines 5–5, at `lecture-12`). Scripts outside the package use absolute imports, such as `from attractor.models import LotkaVolterra`.

1.3 The src layout

A project holds more than its package: scripts, tests, data and configuration. In the **flat layout**, the package folder lies in the root of the project. In the **src layout**, it lies in `src/`. Figure 1 shows the two layouts for the same project.

flat layout

```
attractor/
├── pyproject.toml
├── attractor/
│   ├── __init__.py
│   ├── models.py
│   └── tools.py
└── tests/
    └── test_models.py
```

src layout

```
attractor/
├── pyproject.toml
├── src/
│   └── attractor/
│       ├── __init__.py
│       ├── models.py
│       └── tools.py
└── tests/
    └── test_models.py
```

Figure 1: The same project in the flat layout and in the src layout. In the src layout, the root of the project holds no folder `attractor/` with an `__init__.py`.

Commands are usually run from the root of the project. The root is then the first entry of `sys.path`. The flat layout and the src layout then behave differently (Figure 2).

- With the flat layout, Python finds the package folder in the root, whether the package is installed or not. As a result, a package whose install omits a file still passes its tests in the root.
- With the src layout, the root holds no package folder. As a result, Python imports the installed package. The installed package holds the files that a user receives.

With the src layout, the package must be installed first, for example by the editable install of Section 2.3, or the folder `src/` must be added to `sys.path` by hand. With the editable install, an edit of the source takes effect without a reinstall.

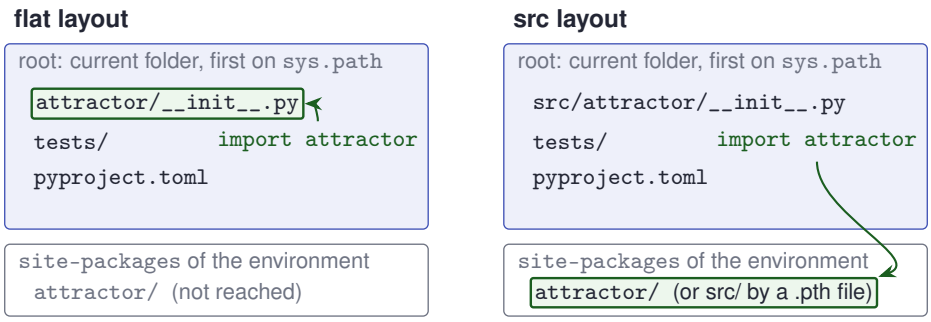


Figure 2: What `import attractor` finds when run in the root of a project. Left: the package folder of the root, installed or not. Right: the root holds no `attractor`, and Python imports the installed package.

2 Project metadata and installation

A program that others install must state its name, its version and the packages it needs. The program states this information in a file that an installer reads. An environment built for the program must also match the environment in which the program was tested.

2.1 Distributions and the project file

Definition 2.1: Distribution and project metadata

A **distribution**, or distribution package, is what pip installs. A distribution is a named and versioned archive. The archive holds modules or import packages, and their **metadata**. The metadata give the name, the version, the interpreters accepted and the distributions required. The name of a distribution and the name imported may differ.

A project declares its metadata in `pyproject.toml`. The file `pyproject.toml` uses the TOML format. A TOML file is made of tables `[name]` and lines `key = value`.

- The table `[build-system]` names the **build backend**. The build backend is the tool that builds the archive from the project. The table `[build-system]` also lists the requirements of the backend.
- The table `[project]` holds the metadata.
- A table `[tool.name]` holds the settings of one tool.

The distribution `python-dotenv` installs the import package `dotenv` (Listing 4). In the same way, `pip install scikit-learn` installs the import package `sklearn`. The metadata of `python-dotenv` give the version 1.2.4. The field `Requires-Python` accepts Python 3.10 and later. The distribution requires `click` for one feature only (Section 2.2).

In TOML, the table `[project.scripts]` is the table `scripts` inside the table `project`. The module `tomllib` of the standard library reads a project file into nested dictionaries (sessions/lecture12/toml.txt, lines 4–16, at lecture-12). For example, in a file with the table `[project.scripts]`, `doc["project"]["scripts"]` is `{'toy': 'toy.cli:main'}`.

```
>>> import importlib.metadata as md
>>> md.packages_distributions()["dotenv"]
['python-dotenv']
>>> meta = md.metadata("python-dotenv")
>>> meta["Version"], meta["Requires-Python"]
('1.2.4', '>=3.10')
>>> md.requires("python-dotenv")
['click>=5.0; extra == "cli"']
>>> (ep,) = md.entry_points(group="console_scripts", name="dotenv")
>>> ep.value, ep.module, ep.attr
('dotenv.__main__:cli', 'dotenv.__main__', 'cli')
```

Listing 4: The distribution `python-dotenv`, its metadata and its entry point: sessions/lecture12/metadata.txt, lines 4–14, at lecture-12.

pip builds a project in a temporary environment. pip first installs the backend into the temporary environment. Building in a temporary environment is called **build isolation**. With build isolation, the build does not depend on the packages of the active environment.

In `attractor`

The project file of `attractor` names setuptools 77 or later as its backend (pyproject.toml, lines 1–3, at lecture-12). The table `[tool.setuptools.packages.find]` tells setuptools to search for packages in `src/` (pyproject.toml, lines 23–24, at lecture-12).

Listing 5 gives the metadata of `attractor`. The text of `README.md` becomes the long description. pip refuses to install the package under Python older than 3.12.

```
[project]
```

```toml
name = "attractor"
version = "1.0.1"
description = "Predator-prey models, fixed-step integrators and cycle periods"
readme = "README.md"
license = "MIT"
requires-python = ">=3.12"
dependencies = ["numpy>=1.26"]

[project.optional-dependencies]
fit = ["scipy>=1.11"]
plot = ["matplotlib>=3.8"]
```

Listing 5: The metadata, the requirement and the two extras: `pyproject.toml`, lines 5–16, at `lecture-12`.

2.2 Version specifiers and optional dependencies

Definition 2.2: Version specifier

A version is a sequence of numbers separated by dots. Two versions are compared component by component, as integers. A missing component counts as 0. For example, 1.10 follows 1.9, and 1.15 equals 1.15.0.

A version may also carry a suffix, such as `rc1` for a release candidate. A release candidate comes before its release. For example, `Version("2.0.0rc1") < Version("2.0.0")` is `True`.

A **version specifier** is a list of clauses separated by commas. A version satisfies the specifier when the version satisfies all the clauses.

- A clause compares with a version by ==, `!=`, <, <=, > or >=.
- The wildcard `==V.*` accepts the series that starts with V. The clause `!=V.*` excludes the series that starts with V.
- The compatible release `~=V` accepts V and the later versions that agree with V in every component but the last. For example, `~=1.11.2` means `>=1.11.2,==1.11.*`. The specifier `~=1.11` means `>=1.11,==1.*`.

Listing 6 applies seven specifiers to five releases of SciPy, with `SpecifierSet` of the library `packaging`. pip uses the library `packaging` to read requirements. Table 2 gives the meaning of each specifier.

The class `Version` of `packaging` confirms the two rules of comparison. For example, `Version("1.10.1") > Version("1.9.3")` is `True`. The comparison `Version("1.15") == Version("1.15.0")` is also `True` (`sessions/lecture12/specifiers.txt`, lines 6–7, at `lecture-12`).

```
>>> releases = ["1.9.3", "1.10.1", "1.11.4", "1.15.3", "1.18.1"]
>>> specs = ["~=1.11.2", "~=1.11", "==1.15", "==1.15.*",
...          "!=1.11.*", ">=1.10,<1.15", "<=1.15"]
>>> for spec in specs:
...     print(f"{spec:<13}", list(SpecifierSet(spec).filter(releases)))
~=1.11.2      ['1.11.4']
~=1.11        ['1.11.4', '1.15.3', '1.18.1']
==1.15        []
==1.15.*      ['1.15.3']
!=1.11.*      ['1.9.3', '1.10.1', '1.15.3', '1.18.1']
>=1.10,<1.15  ['1.10.1', '1.11.4']
```

```
<=1.15        ['1.9.3', '1.10.1', '1.11.4']
```

Listing 6: Seven specifiers on five releases of SciPy: `sessions/lecture12/specifiers.txt`, lines 8–19, at `lecture-12`.

Table 2: The specifiers of Listing 6.

specifier	meaning
`~=1.11.2`	`>=1.11.2,==1.11.*`: later fixes of the 1.11 series
`~=1.11`	`>=1.11,==1.*`: every 1.x from 1.11 on
`==1.15`	exactly 1.15.0, so not 1.15.3
`==1.15.*`	any release of the 1.15 series
`!=1.11.*`	any release outside the 1.11 series
`>=1.10,<1.15`	both clauses: from 1.10.0 up to 1.15.0, excluded
`<=1.15`	at most 1.15.0, so not 1.15.3

An **optional dependency** is a requirement of one feature only. The table `[project.optional-dependencies]` groups optional dependencies under names, called **extras**. The command `pip install "dist[extra]"` installs the requirements of the extra with the distribution.

In the metadata, an optional requirement names its extra. An example is `click>=5.0; extra == "cli"` in Listing 4. Without `click`, the command `dotenv` prints the message `Run pip install "python-dotenv[cli]" to fix this.`

In zsh, the default shell of macOS, square brackets form a pattern of file names. The command `pip install -e .[plot]` therefore stops with `zsh: no matches found: .[plot]`. The quoted form `".[plot]"` works in every shell.

In `attractor`

The modules of `attractor` need NumPy alone (Listing 5). Only the command `attractor-fit` uses SciPy. Only the option `--plot` of `attractor-periods` uses Matplotlib. SciPy and Matplotlib are therefore declared in extras.

`cli.py` imports each of the two libraries inside the function that uses the library (`src/attractor/cli.py`, lines 16–23, at `lecture-12`). Without Matplotlib, the command `attractor-periods --plot` stops. The command prints `matplotlib is not installed: pip install "attractor[plot]"`.

An import inside a function delays the cost of the import until the function runs. Python 3.15 adds explicit lazy imports, which delay an import written at the top of a module until the first use of the name. The change was proposed and accepted as a **Python Enhancement Proposal** (PEP), the document in which a change to Python is proposed, discussed and decided.

2.3 Editable installs and lock files

A regular install copies the files of a distribution into `site-packages`. An **editable install**, `pip install -e .`, works in three steps.

1. pip writes the metadata of the project into `site-packages`.
2. pip writes into `site-packages` a file ending in `.pth`, which names the folder `src/` of the project.
3. At each start-up of Python, the module `site` appends the folder `src/` to `sys.path`.

As a result, an edited module takes effect at the next run. The metadata behave differently. The metadata record the version, the requirements and the console commands of Section 3.1. The metadata change only when pip installs the project again.

A lock file lists every distribution of an environment, with its exact version. The chapter on environments introduced lock files. Suppose that pip installs the project after the lock. Without options, pip then checks the requirements of the project in two steps.

1. pip keeps a locked version that satisfies the requirements.
2. Where no locked version satisfies a requirement, pip installs another version.

In Listing 7, the requirement `numpy<2.5` would replace the locked NumPy 2.5.4 by 2.4.6.

The option `--no-deps` prevents the replacement. With `--no-deps`, pip installs the project alone and leaves the locked versions unchanged. The command `pip check` then reports any missing or incompatible dependency.

```
$ python -m pip install --dry-run "numpy<2.5"
Collecting numpy<2.5
  Downloading numpy-2.4.6-cp313-cp313-macosx_14_0_arm64.whl.metadata (6.6 kB)
Would install numpy-2.4.6
```

Listing 7: A requirement that the locked NumPy 2.5.4 does not satisfy: `requirements-lock.txt` at `lecture-12`.

A published result depends on the code, the data and the software environment that ran them. Reproducing the result later requires the same environment. A lock file records this environment.

In `attractor`

`setup_env.sh` installs the lock first. `setup_env.sh` then installs the project, editable and without dependencies (`setup_env.sh`, lines 61–66, at `lecture-12`). `setup_env.sh` installs the project again whenever `pyproject.toml` is newer than its stamp file.

At `lecture-12-start`, `setup_env.sh` prints the version and the file of the package. The `.pth` file names `src/` (Listing 8). `pip check` then prints `No broken requirements found.`

```
$ source setup_env.sh
setup_env.sh: installing the project (pip install -e . --no-deps)
[...]
attractor       1.0.0
attractor file  ~/GitHub/attractor/src/attractor/__init__.py
[...]
$ cat .venv/lib/python3.13/site-packages/__editable__.attractor-1.0.0.pth
~/GitHub/attractor/src
```

Listing 8: The editable install and its `.pth` file: `setup_env.sh` at `lecture-12-start`.

The command `git switch --detach lecture-12` changes the version in `pyproject.toml` to 1.0.1. An editable install made at `lecture-12-start` still reports `1.0.0` until the project is installed again.

3 Console commands and the public interface

A package is used in two ways. A package provides commands typed at the shell. A package is also a library that other programs import. Both uses rely on names. A later version must

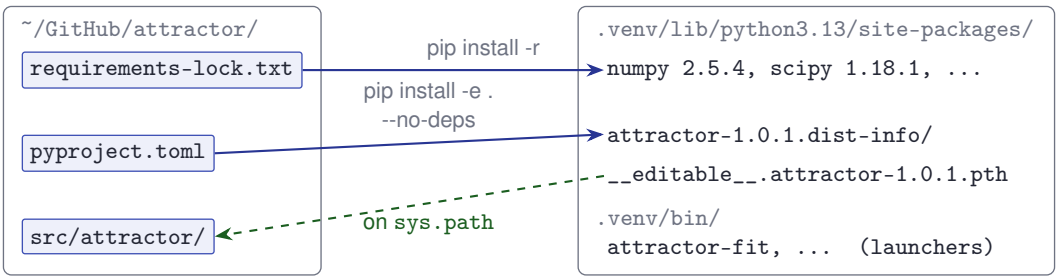


Figure 3: An environment built from the lock and the project file. pip installs the exact versions of the lock. At the editable install, pip writes the metadata, the launchers of the commands and a `.pth` file. The `.pth` file puts the project's `src/` on the import search path.

keep the names of the commands and of the library, or break visibly.

3.1 Console commands

Definition 3.1: Entry point and console command

An **entry point** is a named reference to an object of a package. An entry point is written `module:attribute`. Entry points are stored in groups of the metadata.

Each line `name = "pkg.module:function"` of the table `[project.scripts]` is an entry point of the group `console_scripts`. The entry point makes `name` a **console command**. For example, the line `attractor-fit = "attractor.cli:fit"` creates the command `attractor-fit`, which calls the function `fit` of the module `attractor.cli`.

A console command works in three steps.

1. At install time, pip writes a short script `name` into the folder `bin/` of the environment. This script is the **launcher**.
2. When the command runs, the launcher imports the module and calls the function without arguments. The function reads the command's arguments from `sys.argv`.
3. The launcher passes the return value to `sys.exit`. A return value `None` gives the exit status 0. For example, `sys.exit(2)` gives the exit status 2.

`python-dotenv` declares the entry point `dotenv = dotenv.__main__:cli` (Listing 4). The command `pip` is also a launcher. The entry point of `pip` is `pip._internal.cli.main:main`.

In `attractor`

`attractor` declares three commands (Listing 9). `attractor-simulate` runs a model. `attractor-periods` prints a table of cycle periods. `attractor-fit` fits a model to a table of data.

Each function parses its options with `argparse`. The function reads the options from `sys.argv`, or from a list `argv` (src/attractor/cli.py, lines 26–38, at `lecture-12`). As a result, `simulate(["--h", "0.1"])` can be called from Python as well (sessions/lecture12/package.txt, lines 15–17, at `lecture-12`).

The first line of the launcher names the interpreter of the environment by its absolute path. The package is installed in that environment. So the command runs from any folder (Listing 10).

```
[project.scripts]
attractor-simulate = "attractor.cli:simulate"
```

```
attractor-periods = "attractor.cli:periods"
attractor-fit = "attractor.cli:fit"
```

Listing 9: Three console commands: `pyproject.toml`, lines 18–21, at `lecture-12`.

```
$ cat .venv/bin/attractor-simulate
#![...]/attractor/.venv/bin/python
import sys
from attractor.cli import simulate
if __name__ == '__main__':
    [...]
    sys.exit(simulate())
$ cd / && attractor-simulate
801 points, step 0.05, algae period 8.16 days
```

Listing 10: The launcher written by pip, and the command run from the root of the file system: `src/attractor/cli.py` at `lecture-12-start`.

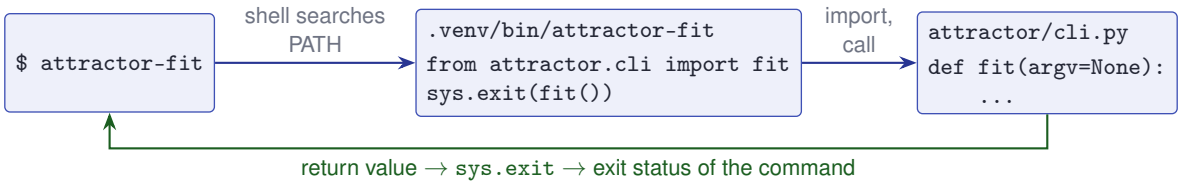


Figure 4: A console command. The shell finds the launcher in the `bin/` folder of the active environment. The launcher imports the function named by the entry point and calls the function. `sys.exit` turns the return value into the exit status.

3.2 The public interface

The **public interface** of a package is the set of names that other code may import from the package and rely on. Names that start with an underscore are private by convention. For example, `json._default_decoder` exists, but a later version of `json` may remove it.

A package lists its public names in `__all__`. The statement `from package import *` binds the names listed in `__all__`.

The `__init__.py` of a package imports the public names from the submodules. Users can then write `package.name`, whichever module defines the name. For example, `json.JSONDecoder is json.decoder.JSONDecoder` is `True`.

The package `json` imports `codecs` for its own use. As a result, `json.codecs` exists. The name `codecs` is not in `json.__all__`. So `from json import *` does not bind `codecs` (Listing 11).

The namespace of a module holds every name bound in the module. The public interface holds only the names that users are meant to rely on. For example, after `from json import *`, the name `codecs` raises `NameError`, while `json.codecs` is the module `codecs`.

```
>>> import json
>>> json.__all__
['dump', 'dumps', 'load', 'loads', 'JSONDecoder', 'JSONDecodeError', 'JSONEncoder']
>>> "codecs" in vars(json), "codecs" in json.__all__
(True, False)
>>> names = {}
>>> exec("from json import *", names)
>>> sorted(name for name in names if name != "__builtins__")
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', 'dump', 'dumps', 'load', 'loads']
```

Listing 11: The public names of `json`: `sessions/lecture12/interface.txt`, lines 4–12, at `lecture-12`.

In `attractor`

The `__init__.py` of `attractor` imports the public names from the modules of the package (Listing 12). The file `__init__.py` does not import `cli`. So `import attractor` loads neither SciPy nor Matplotlib.

The step functions and the decorators can still be imported from their modules. The step functions and the decorators are not public. A later version of `attractor` may change the step functions and the decorators.

The import under the alias `_version` binds no name `version` in the package. `__init__.py` reads `__version__` from the metadata that pip writes at install time (`sessions/lecture12/package.txt`, lines 6–7, at `lecture-12`). So the version is written in one place only, `pyproject.toml`.

```
from importlib.metadata import version as _version

from .cycles import cycle_period, peaks
[...]
from .trajectory import Trajectory

__version__ = _version("attractor")  # from the metadata written at install time

__all__ = ["DynamicalSystem", "LotkaVolterra", "RosenzweigMacArthur", "Goodwin",
           "CHEMOSTAT", "Euler", "RK4", "integrate", "Trajectory", "peaks",
           "cycle_period", "load_run", "lynx_hare"]
```

Listing 12: The public interface of `attractor`: `src/attractor/__init__.py`, lines 5–18, at `lecture-12`.

3.3 Semantic versions

Definition 3.2: Semantic version

A **semantic version** `MAJOR.MINOR.PATCH` states what a release does to the public interface.

- A release that breaks code written for the previous one increments MAJOR.
- A release that adds to the interface and breaks nothing increments MINOR.
- A release that only fixes faults increments PATCH.

The components to the right of the one incremented reset to 0. So 2.3.1 is followed by 2.3.2, 2.4.0 or 3.0.0. A major version 0 indicates an interface that is not yet stable. A published version is never reused for other files.

NumPy 2.0.0 removed the alias `np.NaN`. An analysis script of the chapter on environments was written for NumPy 1.26. Under NumPy 2, the script stopped with an `AttributeError`. For example, under NumPy 2.5.4, `np.NaN` raises `AttributeError: ‘np.NaN‘ was removed in the NumPy 2.0 release`.

The major release broke code that the 1.x series ran. NumPy 1.26.4 is the fourth release of fixes to 1.26. The requirement `~=1.26` relies on the convention of semantic versions.

The requirement `~=1.26` accepts every 1.x from 1.26 on. The requirement accepts no 2.x. For

example, `"2.0.0" in SpecifierSet("~=1.26")` is `False`.

In `attractor`

The `__all__` of `attractor` defines the public interface of the package. As a result, the first version of `attractor` is 1.0.0. The fix of Section 4.2 changes no public name. The fixed release is therefore 1.0.1 (Listing 17).

4 Wheels

An installed package is a set of files copied into an environment from an archive. The archive is built before the install. The build of the archive decides which files the archive holds. The build also decides which interpreters can use the files.

4.1 The contents of a wheel

Definition 4.1: Wheel

A **wheel** is a zip archive named `name-version-pythontag-abitag-platformtag.whl`. pip unpacks a wheel into `site-packages` without running code of the project. A wheel holds three kinds of files.

- The import packages that the backend finds, in `src/` for the src layout.
- The files declared as package data (Section 4.2).
- A folder `name-version.dist-info`. The file `METADATA` holds the fields of `[project]`, then the text of the readme file. `WHEEL` names the builder and the tags. `RECORD` lists every file, with its hash and size. `entry_points.txt` lists the console commands.

A wheel does not hold tests, documentation, scripts outside the package or the readme file itself.

An installed distribution keeps the `dist-info` folder of its wheel. For `python-dotenv` (`sessions/lecture12/wheel.txt, lines 4–13, at lecture-12`), the file `entry_points.txt` declares the command `dotenv`. The file `WHEEL` reads `Root-Is-Purelib: true` and `Tag: py3-none-any`. So the wheel of `python-dotenv` holds Python code alone.

A wheel of NumPy also holds compiled extension modules, files ending in `.so`. The tags of a NumPy wheel name the interpreters that can load the compiled modules (Section 4.3).

In `attractor`

The command `python -m pip wheel . --no-deps -w dist` builds the wheel of the project in the current folder. It writes the wheel into `dist/`. With `--no-deps`, pip does not build the wheels of the dependencies of the project.

At `lecture-12-start`, the wheel of version 1.0.0 holds the eight modules and the metadata (Listing 13). The file `METADATA` of this wheel starts with `Name: attractor` and `Version: 1.0.0`.

```
$ python -m pip wheel . --no-deps -w dist -q
$ unzip -Z1 dist/attractor-1.0.0-py3-none-any.whl
attractor/__init__.py
attractor/cli.py
[...]
attractor/trajectory.py
```

```
attractor-1.0.0.dist-info/licenses/LICENSE
attractor-1.0.0.dist-info/METADATA
attractor-1.0.0.dist-info/WHEEL
attractor-1.0.0.dist-info/entry_points.txt
attractor-1.0.0.dist-info/top_level.txt
attractor-1.0.0.dist-info/RECORD
```

Listing 13: The wheel of version 1.0.0: `pyproject.toml` at `lecture-12-start`.

4.2 Package data

Package data are the files of a package that are not modules, such as tables and default settings. setuptools treats modules and data files differently.

- setuptools puts a module into the wheel because the module lies in a package.
- setuptools puts a data file into the wheel only if the data file is declared in the table `[tool.setuptools.package-data]`.

Each key of the table `[tool.setuptools.package-data]` names a package. Each value lists glob patterns, such as `data/*.csv`. A glob pattern matches file names. Each pattern is relative to the folder of the package. For example, `PurePath("data/lynx_hare.csv").match("data/*.csv")` is `True`.

The code of a package reads a data file through `importlib.resources.files(package)`. The object returned by `files(package)` represents the folder of the package, wherever the package is installed. Listing 14 reads the default settings of Matplotlib, `mpl-data/matplotlibrc`, through `files("matplotlib")`. For a path that the package lacks, `is_file` returns `False`.

```
>>> from importlib.resources import files
>>> rc = files("matplotlib").joinpath("mpl-data/matplotlibrc")
>>> rc.is_file(), rc.read_text().splitlines()[0]
(True, '#### MATPLOTLIBRC FORMAT')
>>> files("matplotlib").joinpath("mpl-data/no_such_file").is_file()
False
```

Listing 14: A file of Matplotlib's package data: `sessions/lecture12/resources.txt`, lines 4–9, at `lecture-12`.

In `attractor`

The package `attractor` holds a table, `data/lynx_hare.csv`. The function `lynx_hare` reads the table (Listing 15). The command `attractor-fit` calls `lynx_hare`. The fix of the missing table takes three steps.

1. At `lecture-12-start`, nothing declares the table. The wheel of 1.0.0 lacks the table. As a result, `attractor-fit` fails in a new environment (Listing 16). With the editable install, `attractor-fit` runs without error, since `lynx_hare` reads the file from `src/`.
2. At `lecture-12`, two lines declare the table. The version becomes 1.0.1 (Listing 17).
3. The new wheel holds the table. In the new environment, `attractor-fit` prints `LotkaVolterra(a=0.481, b=0.0248, c=0.0275, d=0.926)`.

```
def lynx_hare():
    """Years 1900-1920, lynx and hare pelts (thousands), from the package."""
```

```
    text = files("attractor").joinpath("data/lynx_hare.csv").read_text()
    d = np.genfromtxt(text.splitlines(), delimiter=",", skip_header=3)
    return d[:, 0], d[:, 1], d[:, 2]
```

Listing 15: The reader of the table in the package: `src/attractor/datasets.py`, lines 16–20, at `lecture-12`.

```
$ python3.13 -m venv ../fresh
$ ../fresh/bin/python -m pip install -q "dist/attractor-1.0.0-py3-none-any.whl[fit]"
$ ../fresh/bin/attractor-fit
Traceback (most recent call last):
  File "~/GitHub/attractor/../fresh/bin/attractor-fit", line 7, in <module>
    sys.exit(fit())
[...]
  File "~/GitHub/fresh/lib/python3.13/site-packages/attractor/datasets.py", line 18, in lynx_hare
    text = files("attractor").joinpath("data/lynx_hare.csv").read_text()
[...]
FileNotFoundError: [Errno 2] No such file or directory: '[...]/site-packages/attractor/data/
    lynx_hare.csv'
```

Listing 16: The wheel of 1.0.0 in a new environment: `src/attractor/datasets.py` at `lecture-12-start`.

```
$ git diff lecture-12-start lecture-12
[...]
-version = "1.0.0"
+version = "1.0.1"
[...]
+[tool.setuptools.package-data]
+attractor = ["data/*.csv"]
$ python -m pip wheel . --no-deps -w dist -q
$ unzip -Z1 dist/attractor-1.0.1-py3-none-any.whl | grep csv
attractor/data/lynx_hare.csv
```

Listing 17: The fix and the new wheel: `lecture-12-start` to `lecture-12`.

An editable install reads package data from `src/`. The file lies in `src/`, declared or not. As a result, the editable install hides a missing declaration. Only a wheel installed into a new environment reveals the missing declaration.

In a project tree built before, setuptools keeps an undeclared file in later wheels. setuptools reads the list of files of the earlier build, `src/attractor.egg-info/SOURCES.txt`. A fresh clone, or the deletion of `build/` and `src/*.egg-info`, removes the stale list.

4.3 Compatibility tags

The last three fields of a wheel's name are its **compatibility tags** (Figure 5).

- The Python tag names the interpreters: `py3` for any Python 3, `cp313` for CPython 3.13.
- The ABI tag names the **application binary interface**. The ABI is the set of conventions that compiled code relies on when calling the interpreter. The tag `none` means no compiled code. `cp313` means code compiled against CPython 3.13. `abi3` means the stable ABI, a subset that later CPython 3 versions keep.
- The platform tag names the operating system, its oldest supported release and the processor. For example, `macosx_14_0_arm64` names macOS 14.0 or later on arm64. `universal2` means arm64 and x86_64 both. `any` means any platform.

The tag `py3-none-any` marks a wheel of Python code alone. The wheel runs on any Python 3, needs no particular ABI, and runs on any platform.

pip chooses a wheel in three steps.

1. The interpreter accepts an ordered list of tags, from the most specific down to `py3-none-any`. The command `python -m pip debug --verbose` prints the list.
2. pip installs a wheel only if one of its tags is in the list.
3. Among several compatible wheels, pip prefers the one whose tag comes first.

Listing 18 reads the tag of the installed NumPy 2.5.4. The NumPy wheel was compiled for CPython 3.13 on macOS 14 or later. Listing 18 then prints the first tags that CPython 3.13 accepts on the lecture laptop. CPython 3.13 accepts 683 tags there, on macOS 15 with an arm64 processor.

```
$ cat .venv/lib/python3.13/site-packages/numpy-2.5.4.dist-info/WHEEL
Wheel-Version: 1.0
Generator: meson
Root-Is-Purelib: false
Tag: cp313-cp313-macosx_14_0_arm64
[...]
$ python -m pip debug --verbose 2>/dev/null | grep -A3 "Compatible tags"
Compatible tags: 683
  cp313-cp313-macosx_15_0_arm64
  cp313-cp313-macosx_15_0_universal2
  cp313-cp313-macosx_14_0_arm64
```

Listing 18: The tag of a compiled wheel, and the tags that an interpreter accepts: `requirements-lock.txt` at `lecture-12`.

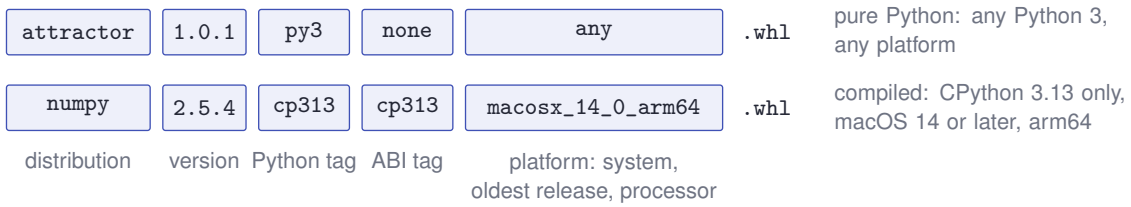


Figure 5: The fields of two wheel names: the project's wheel, and the NumPy wheel of the course environment.

In `attractor`

The wheel of `attractor` is tagged `py3-none-any` (Listing 19). In the Python 3.13 environment of the project, pip accepts the wheel. In a Python 3.10 environment, pip refuses the wheel because of `requires-python`. In the Python 3.13 environment, pip refuses a NumPy 2.5.4 wheel compiled for CPython 3.12, tagged `cp312-cp312-macosx_14_0_arm64`.

```
$ python -m pip install --dry-run ../wheels/numpy-2.5.4-cp312-cp312-macosx_14_0_arm64.whl
ERROR: numpy-2.5.4-cp312-cp312-macosx_14_0_arm64.whl is not a supported wheel on this platform.
$ python -m pip install --dry-run --no-deps dist/attractor-1.0.1-py3-none-any.whl
Processing ./dist/attractor-1.0.1-py3-none-any.whl
Would install attractor-1.0.1
$ ../old/bin/python -m pip install --dry-run --no-deps dist/attractor-1.0.1-py3-none-any.whl
Processing ./dist/attractor-1.0.1-py3-none-any.whl
[...]
ERROR: Package 'attractor' requires a different Python: 3.10.18 not in '>=3.12'
```

Listing 19: Three wheels offered to Python 3.13, and one to Python 3.10: `pyproject.toml` at `lecture-12`.

Scientific software is useful to others only if it installs on their machines. The tags of a wheel decide which machines can install the package.

Troubleshooting

symptom	cause	remedy
`ModuleNotFoundError: No module named 'attractor'`	The interpreter that runs is not that of the environment holding the project (Definition 1.1).	`source setup_env.sh`, or `pip install -e .` in the active environment.
`zsh: command not found: attractor-fit`	The environment is not active, so its `bin/` is not on `PATH`. Or the command was declared after the install (Definition 3.1).	Activate the environment, then install the project again.
`ERROR: [...] is not a supported wheel on this platform.`	The interpreter accepts none of the wheel's tags (Section 4.3).	The wheel built for that interpreter.
`pip check` reports a missing requirement	The project declares a requirement that the lock lacks. With `--no-deps`, pip did not install the requirement (Section 2.3).	Add the requirement to the specification and regenerate the lock.

Table 3: Files, commands and functions of this chapter.

object	effect	section
`sys.path`, `importlib.util.find_spec`	the folders that `import` searches, and the search	1.1
`__init__.py`, `from .mod import f`	a regular package, and a relative import	1.2
`[build-system]`, `[project]`, `packages.find`	the backend, the metadata, where the packages lie	1.3, 2.1
`[project.optional-dependencies]`, `pip install -e ".[plot]"`	named extras, and an install with one	2.2
`pip install -e . --no-deps`, `pip check`	the project alone, editable, and the check of its requirements	2.3
`[project.scripts]`, `__all__`	console commands, and the public names	3.1, 3.2
`pip wheel . --no-deps -w dist`, `pip debug --verbose`	the wheel of the project, and the tags the interpreter accepts	4.1, 4.3
`[tool.setuptools.package-data]`, `importlib.resources.files`	declared data files, and the folder of an installed package	4.2

Key points

- Python imports the first match along `sys.path`. The first entry is the script's folder or the current folder. So a package in `src/` is found only once it is installed (Definition 1.1, Figure 2).
- A version must satisfy every clause of a specifier. Versions compare number by number. `~=V` accepts V and the later versions that agree with V in every component but the last (Definition 2.2, Table 2).
- With an editable install, code edits take effect at once. Metadata edits take effect only at a reinstall. After a lock, `pip install -e . --no-deps` keeps the locked versions (Section 2.3).
- A wheel holds the packages, the declared package data and the `dist-info` files. An undeclared data file is missing after a regular install. The tags of a wheel name the interpreters that can install the wheel (Definition 4.1, Figure 5).

Exercises

Solutions are given from page 21, checked by the scripts of `exercises/lecture12/`.

Requirements on python-dotenv

Consider five requirements on `python-dotenv`: `~=1.0.1`, `~=1.0`, `==1.*`, `!=1.0.*` and `>0.21,<=1.1`. (a) Which of the releases 0.21.1, 1.0.0, 1.0.1, 1.1.0 and 1.2.4 does each requirement accept? (b) Which requirements accept the later releases 1.10.0 and 2.0.0? (c) Which two requirements agree in (a) and (b)?

Contents and tags of wheels

(a) The first four files on the left below are added to the project at `lecture-12`. Then the wheel of the project is built. Which files on the left does the wheel hold? (b) Which wheels on the right install into CPython 3.13 on macOS 15 and arm64? Which wheels install into CPython 3.12 on macOS 13 and x86_64?

```
src/attractor/data/C0.csv
src/attractor/data/notes.txt
tests/test_cycles.py, docs/index.md
src/attractor/datasets.py, data/C9.csv
simulate.py, README.md
```

```
attractor-1.0.1-py3-none-any.whl
tornado-6.5.10-cp39-abi3-macosx_10_9_universal2.whl
psutil-7.2.2-cp36-abi3-macosx_11_0_arm64.whl
debugpy-1.8.22-cp312-cp312-macosx_15_0_universal2.whl
```

A module run as a file

A folder holds the files of Listing 20. Each key is a file name. Five commands are run in this folder: `python geometry/shapes.py`, `python -m geometry.shapes`, `python scripts/area.py`, `python -m scripts.area` and `cd scripts && python -m area`. Give the last line printed by each command.

For each failure, give the entry of `sys.path` that explains the failure.

```
FILES = {
    "geometry/__init__.py": "",
    "geometry/units.py": "MM_PER_CM = 10\n",
    "geometry/shapes.py": "from .units import MM_PER_CM\n"
                          "print('30 cm =', 30 * MM_PER_CM, 'mm')\n",
    "scripts/area.py": "import geometry.shapes\n",
}
```

Listing 20: The four files: `exercises/lecture12/e3_relative.py`, lines 6–12, at `lecture-12`.

The cost of an import

The command `python -X importtime -c "import attractor"` prints the time of each import. Its second column is the cumulative time, in microseconds.

(a) Measure the import times of `attractor` and `numpy`. Then measure the import times of `attractor.cli` and `scipy.optimize`. (b) Suppose `cli.py` imported SciPy at its top. Estimate what `attractor-simulate` would then cost. Explain why `import attractor` imports no SciPy.

Metadata written at install time

The project at `lecture-12` is installed editable. Then three edits are made.

- `pyproject.toml` gets `version = "1.1.0"`.
- `pyproject.toml` gets a command in `[project.scripts]`: `attractor-version = "attractor.cli:show_version"`.
- `cli.py` gets a function `show_version` that prints `attractor` and `__version__`.

(a) Without a reinstall, what do `print(attractor.__version__)`, `show_version()` and `attractor-version` print? (b) What do they print after `pip install -e . --no-deps`? (c) Which edit took effect at once, and why?

Solutions to the exercises

Requirements on python-dotenv

(a) `~=1.0.1` means `>=1.0.1,==1.0.*`. The specifier `~=1.0.1` accepts 1.0.1. `~=1.0` means `>=1.0,==1.*`. The specifier `~=1.0` accepts 1.0.0, 1.0.1, 1.1.0 and 1.2.4. `==1.*` also accepts 1.0.0, 1.0.1, 1.1.0 and 1.2.4.

`!=1.0.*` accepts 0.21.1, 1.1.0 and 1.2.4. `>0.21,<=1.1` accepts 0.21.1, 1.0.0, 1.0.1 and 1.1.0, since 1.1 equals 1.1.0.

(b) The second component of 1.10.0 is 10, which exceeds 1. So 1.10.0 is accepted by `~=1.0`, `==1.*` and `!=1.0.*`. 2.0.0 is accepted by `!=1.0.*` alone.

(c) `~=1.0` and `==1.*` agree. For final releases, `~=1.0` and `==1.*` both mean the 1.x series.

Answer check

```
$ python exercises/lecture12/e1_specifiers.py
~=1.0.1      ['1.0.1']  later: []
~=1.0        ['1.0.0', '1.0.1', '1.1.0', '1.2.4']  later: ['1.10.0']
==1.*        ['1.0.0', '1.0.1', '1.1.0', '1.2.4']  later: ['1.10.0']
!=1.0.*      ['0.21.1', '1.1.0', '1.2.4']  later: ['1.10.0', '2.0.0']
>0.21,<=1.1  ['0.21.1', '1.0.0', '1.0.1', '1.1.0']  later: []
```

Listing 21: The five requirements on the releases, then on the two later ones: `exercises/lecture12/e1_specifiers.py` at `lecture-12`.

Contents and tags of wheels

(a) The wheel holds two of the files. `C0.csv` matches `data/*.csv`. `datasets.py` is a module of the package. The wheel does not hold the other six files.

- `notes.txt`: the glob `data/*.csv` does not match `notes.txt`.
- The test, the documentation, `data/C9.csv` and `simulate.py`: these files lie outside the package.
- `README.md`: its text is in `METADATA`.

(b) CPython 3.13 on arm64 installs `attractor`, `tornado` and `psutil`. `tornado` uses the stable ABI from 3.9, for both processors. `psutil` uses the stable ABI from 3.6, on arm64. CPython 3.13 refuses `debugpy`, which was compiled for CPython 3.12.

CPython 3.12 on an x86_64 Mac under macOS 13 installs `attractor` and `tornado`. CPython 3.12 refuses `psutil`, which is for arm64 only. CPython 3.12 refuses `debugpy`, which needs macOS 15.

Answer check

```
$ python exercises/lecture12/e2_wheel.py
attractor-1.0.1-py3-none-any.whl
src/attractor/data/C0.csv      in
src/attractor/data/notes.txt   not in
tests/test_cycles.py           not in
docs/index.md                  not in
src/attractor/datasets.py      in
data/C9.csv                    not in
simulate.py                    not in
README.md                      not in
CPython 3.13, macOS 15, arm64: ['attractor', 'tornado', 'psutil']
CPython 3.12, macOS 13, x86_64: ['attractor', 'tornado']
```

Listing 22: The files of the wheel, and the wheels that two interpreters accept, their tags computed with `packaging.tags`: `exercises/lecture12/e2_wheel.py` at `lecture-12`.

A module run as a file

1. The first command runs `shapes.py` outside any package. As a result, Python raises `ImportError` at the relative import of `shapes.py`.
2. The second command imports the package `geometry` from the current folder. The current folder is the first entry of `sys.path`.
3. The third command puts the script's folder, `scripts/`, first. The folder `scripts/` holds no `geometry`.
4. The fourth command puts the current folder first again.
5. The fifth command runs in `scripts/`. As a result, the current folder holds no `geometry`.

Answer check

```
$ python exercises/lecture12/e3_relative.py
python geometry/shapes.py      ImportError: attempted relative import with no known parent package
python -m geometry.shapes      30 cm = 300 mm
python scripts/area.py         ModuleNotFoundError: No module named 'geometry'
python -m scripts.area         30 cm = 300 mm
cd scripts && python -m area   ModuleNotFoundError: No module named 'geometry'
```

Listing 23: The five commands, run in a temporary folder: `exercises/lecture12/e3_relative.py` at `lecture-12`.

The cost of an import

(a) On the lecture laptop, with Python 3.13 and NumPy 2.5, importing `attractor` took 42 ms. NumPy took 28 ms of the 42 ms. `attractor.cli` added 3 ms. `scipy.optimize` alone took 189 ms.

(b) A SciPy import at the top of `cli.py` would add about 0.15 to 0.19 s to every command. The extra time would also apply to `attractor-simulate`, which needs no SciPy. `import attractor` imports no SciPy because `__init__.py` does not import `cli`. `__init__.py` imports only the modules of the library, which need NumPy alone.

Answer check

```
$ python exercises/lecture12/e4_import_cost.py
import numpy             27.9 ms   scipy imported: False
import attractor         42.4 ms   scipy imported: False
import attractor.cli     45.0 ms   scipy imported: False
import scipy.optimize   188.7 ms   scipy imported: True
```

Listing 24: Cumulative import times on the lecture laptop, Python 3.13: `exercises/lecture12/e4_import_cost.py` at `lecture-12`.

Metadata written at install time

(a) `1.0.1`, `attractor 1.0.1`, and no command. (b) `1.1.0`, `attractor 1.1.0` and `attractor 1.1.0`. (c) The new function took effect at once.

With the editable install, Python imports `cli.py` from `src/`. As a result, the edited file runs at once.

The version comes from the metadata. The launcher of the new command lies in `bin/`. pip writes the metadata and the launcher at install time.

Answer check

```
$ python exercises/lecture12/e5_stale_metadata.py
before           import attractor     1.0.1
before           cli.show_version()   attractor 1.0.1
before           attractor-version    command not found
after reinstall  import attractor     1.1.0
after reinstall  cli.show_version()   attractor 1.1.0
after reinstall  attractor-version    attractor 1.1.0
```

Listing 25: A copy of the project installed, edited and installed again: `exercises/lecture12/e5_stale_metadata.py` at `lecture-12`.

References

Boris Bolliet. `attractor`: cycle periods of the chemostat runs. GitHub, https://github.com/borisbolliet/attractor. tags `lecture-12-start` and `lecture-12`.

Python Software Foundation. The import system. Python 3.13 language reference, https://docs.python.org/3/reference/import.html. accessed 25 October 2026.

Python Software Foundation. The initialization of the `sys.path` module search path. Python 3.13 documentation, https://docs.python.org/3/library/sys_path_init.html. accessed 25 October 2026.

Python Packaging Authority. src layout vs flat layout. Python Packaging User Guide, https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/. accessed 25 October 2026.

Python Packaging Authority. `pyproject.toml` specification. Python Packaging User Guide, https://packaging.python.org/en/latest/specifications/pyproject-toml/. accessed 25 October 2026.

Python Packaging Authority. Distribution package vs. import package. Python Packaging User Guide, https://packaging.python.org/en/latest/discussions/distribution-package-vs-import-package/. accessed 25 October 2026.

Python Packaging Authority. Version specifiers. Python Packaging User Guide, https://packaging.python.org/en/latest/specifications/version-specifiers/. originally PEP 440; accessed 25 October 2026.

The pip developers. `pip install`. pip documentation, https://pip.pypa.io/en/stable/cli/pip_install/. options `-e`, `--no-deps` and `--dry-run`; accessed 25 October 2026.

Python Packaging Authority. Development mode (a.k.a. “editable installs”). setuptools documentation, https://setuptools.pypa.io/en/latest/userguide/development_mode.html. accessed 25 October 2026.

Python Packaging Authority. Entry points specification. Python Packaging User Guide, https://packaging.python.org/en/latest/specifications/entry-points/. accessed 25 October 2026.

Python Software Foundation. `importlib.metadata`: accessing package metadata. Python 3.13 documentation, https://docs.python.org/3/library/importlib.metadata.html. accessed 25 October 2026.

Tom Preston-Werner. Semantic Versioning 2.0.0. https://semver.org/. accessed 25 October 2026.

NumPy developers. NumPy 2.0 migration guide. NumPy documentation, https://numpy.org/doc/stable/numpy_2_0_migration_guide.html. accessed 25 October 2026.

Python Packaging Authority. Binary distribution format. Python Packaging User Guide, https://packaging.python.org/en/latest/specifications/binary-distribution-format/. originally PEP 427; accessed 25 October 2026.

Python Packaging Authority. Core metadata specifications. Python Packaging User Guide, https://packaging.python.org/en/latest/specifications/core-metadata/. accessed 25 October 2026.

Python Packaging Authority. Data files support. setuptools documentation, https://setuptools.pypa.io/en/latest/userguide/datafiles.html. accessed 25 October 2026.

Python Software Foundation. `importlib.resources`: package resource reading, opening and access. Python 3.13 documentation, https://docs.python.org/3/library/importlib.resources.html. accessed 25 October 2026.

Python Packaging Authority. Platform compatibility tags. Python Packaging User Guide, https://packaging.python.org/en/latest/specifications/platform-compatibility-tags/. originally PEP 425; accessed 25 October 2026.

Python Software Foundation. Lazy imports. The Python Language Reference 3.15, simple statements, https://docs.python.org/3.15/reference/simple_stmts.html#lazy-imports. accessed 3 November 2026.

Python Software Foundation. PEP 0 – index of python enhancement proposals. https://peps.python.org/. accessed 3 November 2026.

Index

absolute import, 3
application binary interface, 15

build backend, 6
build isolation, 6

compatibility tags, 15
console command, 10

distribution, 6

editable install, 8
entry point, 10
extras, 8

flat layout, 5

import search path, 2

launcher, 10

metadata, 6

optional dependency, 8

package, 3
package data, 14
public interface, 11
Python Enhancement Proposal, 8

relative import, 3

semantic version, 12
src layout, 5
submodules, 3
subpackages, 3

version specifier, 7

wheel, 13