Lecture 4 — Reproducible Python environments

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

Contents

1 Layers of an environment **2**
- 1.1 The three layers of an environment . . . 2
- 1.2 Virtual environments . . . 2
- 1.3 Specification and snapshot . . . 3

2 Package versions **5**
- 2.1 Version specifiers . . . 5
- 2.2 Breaking releases and interpreter-bound wheels . . . 6

3 Configuration outside the code **7**
- 3.1 Configuration in environment variables . . . 7
- 3.2 The .env file and python-dotenv . . . 10
- 3.3 A service address in the configuration . . . 11

4 Rebuilding the environment **11**
- 4.1 The setup script . . . 12
- 4.2 The import search path . . . 14
- 4.3 Jupyter kernels . . . 16
- 4.4 uv, conda and Colab . . . 16

Troubleshooting **19**

Key points **19**

Course environment **20**

Exercises **21**

Solutions to the exercises **22**

The `attractor` project [1] measures the cycle period of ten chemostat runs, `C1.csv` to `C10.csv`, from the peaks of the algae density [2]. Its script `analyse.py` was written in 2024 against NumPy 1.26. Installed afresh in 2026, the same script stops on its first array with `AttributeError: ‘np.NaN‘ was removed in the NumPy 2.0 release`. Nothing in the script changed. The packages around it did. A second copy of the project, on a Linux workstation, must print the same table of periods as the laptop copy, and a notebook opened a month later must still import the packages the scripts need. The chapter is about the files that make this possible: a record of the interpreter, of the installed packages and of the run-time configuration, committed next to the code, and a script that rebuilds a working shell from them on any machine.

The outputs shown were produced on the lecture laptop, a MacBook with an Apple M4 Max processor running macOS, on 20 October 2026, with Python 3.12.12 and Python 3.13.11 from Homebrew, pip 25.2 and a warm pip cache. They were captured in a fresh clone of the repository, whose directory is printed as `/Users/boris/GitHub/attractor`. Two listings are retained

from the lecture of 15 October 2026 and say so: the run on the workstation, with Python 3.12.3 on Ubuntu 24.04, and the reply of the language-model endpoint. Where a printed form depends on a version, the text says which.

1 Layers of an environment

1.1 The three layers of an environment

Definition 1.1: Environment of a project

The **environment** of a Python project is everything outside its source code that its results depend on. It has three layers: the **interpreter**, the `python` executable and its version, the **packages**, the set of distributions installed for that interpreter with their versions, and the **configuration**, the values read at run time that differ between machines and runs, such as data and output directories, thread counts and service addresses. A project is **reproducible** when each layer is recorded in a committed file and can be rebuilt from that file on another machine.

The layers differ in how they are recorded and rebuilt (Table 1, Figure 1). The interpreter is named by the setup script, which creates the virtual environment with `python3.12`, and by a comment in the package snapshot. The packages are recorded twice, as a specification of what the code needs and as a snapshot of what was installed. The configuration is recorded as a template, `.env.example`, whose local copy `.env` holds the values of the machine at hand. The rebuilt objects, the directory `.venv`, the file `.env` and the results, never enter the repository.

Table 1: The three layers of the `attractor` environment, the committed file that records each and the command that rebuilds it.

layer	recorded in	rebuilt by
interpreter	`setup_env.sh` names `python3.12`, and the header of `requirements-lock.txt` names 3.12.12	`python3.12 -m venv .venv`
packages	`requirements.txt`, the specification, and `requirements-lock.txt`, the `pip freeze` snapshot	`python -m pip install -r requirements-lock.txt`
configuration	`.env.example`, a committed template, and `.env`, the local copy, ignored by git	`set -a; source .env; set +a`, or `load_dotenv()` in the script
all three	`setup_env.sh`	`source setup_env.sh`

Listing 1 shows the repository after a clone and after one run of the analysis. The specification `requirements.txt` names the four packages the scripts import, with lower bounds for the two whose interface the code depends on, and `ipykernel` for the notebook kernel. The ignore file keeps every build product out of git: `.venv*/` matches the virtual environment and the second one built in Section 2.2, `results*/` matches every results directory, and `.env` is the local configuration. The directory `__pycache__/` appears once `period.py` has imported `analyse`, since Python caches the compiled module there.

1.2 Virtual environments

A **virtual environment** is a directory holding a `python` executable bound to one interpreter and a private `site-packages` directory for the packages installed with it [3]. It is created once per project with `python3.12 -m venv .venv`, and the interpreter named on that command line is

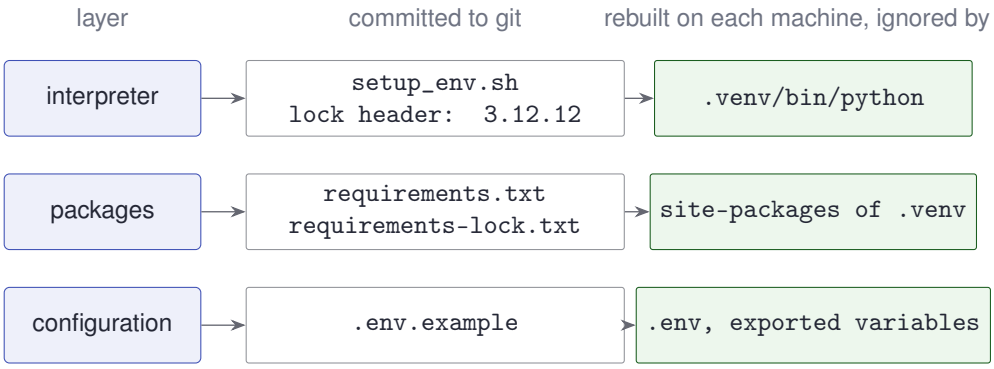


Figure 1: The three layers of Definition 1.1. The middle column is committed with the code. The right column is a build product, listed in `.gitignore` and created again on every machine by `source setup_env.sh`.

the one the environment uses for ever after. Activation, `source .venv/bin/activate`, prepends `.venv/bin` to the `PATH` of the current shell, so that the commands `python` and `pip` resolve to the copies inside the environment. `deactivate` restores the previous `PATH`. Listing 2 shows the three states. Packages are installed with `python -m pip install`, which guarantees that pip installs into the interpreter that will run the code, whatever `pip` alone would resolve to.

> The shell, not the interpreter or the kernel, searches `PATH` [4]. After activation the shell finds `.venv/bin/python` before `/opt/homebrew/bin/python3` or `/usr/bin/python3`, and `which python` prints the file that will run. A notebook kernel is a separate process and does not see this `PATH` change (Section 4.3).

The environment is a build product. With the packages of this project it occupies about 250 MB, measured with `du -sh .venv`, it contains compiled extension modules for one operating system and processor, and it is recreated in seconds from the committed files. It is never committed, which is what the first line of `.gitignore` enforces.

1.3 Specification and snapshot

Definition 1.2: Specification and snapshot of the packages

A **specification** lists the packages the code imports, each with the loosest constraint under which the code is known to work, such as `numpy>=1.26`. It states what the code needs and leaves the choice of versions to the installer. A **snapshot**, also called a **lock file**, lists every distribution installed in one working environment with its exact version, `numpy==1.26.4`, including the packages pulled in as dependencies that the code never names. It states what was actually run. `pip freeze` prints the snapshot of the active environment [5].

The two files answer two questions. The specification answers "what must be present for the code to run", and it is what a new project starts from. The snapshot answers "which exact versions produced these results", and it is what a second machine installs when it must reproduce them. Listing 3 installs the specification into the environment of Listing 2. pip resolves the five requirements to 52 distributions, because `openai` alone brings `httpx2`, `httpcore2`, `pydantic`, `anyio` and `jiter`, and `ipykernel` brings `ipython`, `jupyter_client`, `tornado` and `pyzmq`. The versions are the newest that satisfy the constraints on the day of the install, 20 October 2026: NumPy 2.5.4, Matplotlib 3.11.2, openai 3.35.1. On another day the same command installs other versions.

```
$ git clone https://github.com/borisbolliet/attractor.git ~/GitHub/attractor
$ cd ~/GitHub/attractor
$ ls -a
.                   analyse.py          period.py
..                  ask_llm.py          README.md
.env.example        check_env.py        requirements-lock.txt
.git                data                requirements.txt
.gitignore          LICENSE             setup_env.sh
$ cat requirements.txt
# Specification: lower bounds for imported packages,
# plus ipykernel for the notebook kernel.
numpy>=1.26
matplotlib>=3.8
python-dotenv
openai
ipykernel
$ cat .gitignore
.venv*/
.env
results*/
__pycache__/
.ipynb_checkpoints/
$ git status --ignored --short
!! .env
!! .venv/
!! __pycache__/
!! results/
```

Listing 1: A fresh clone of the project: the committed files, the specification of the packages, and the build products that `git status --ignored` marks with `!!` after the virtual environment has been created and the analysis has run.

> Creating the environment with the interpreter of the original run does not reproduce it. With Python 3.12, `pip install -r requirements.txt` resolves `numpy>=1.26` to NumPy 2.5.4 exactly as it does with Python 3.13, and `analyse.py` fails in both. The 2024 run is reproduced by the snapshot, which pins `numpy==1.26.4`, with an interpreter for which that release has a wheel (Section 2.2).

The committed snapshot, `requirements-lock.txt`, is the `pip freeze` output of the environment in which the scripts were last run, pinned with == on every line (Listing 4). The environment of Listing 3 rebuilt from it prints a `pip freeze` identical to the file, 52 lines. Three lines of comment precede the list: the interpreter, Python 3.12.12, the platform, macOS on arm64, the date, and the command that produced the environment, which installed the specification with two extra constraints, `numpy==1.26.4` and `matplotlib<3.10`. pip writes none of this.

> `pip freeze` records the versions of the installed packages and nothing else. It does not record the interpreter version, the operating system or the processor architecture, and every pinned release needs a wheel for all three (Definition 2.2). A snapshot without a header stating them cannot be installed with confidence on another machine. Entries such as `appnope`, a macOS-only dependency of `ipykernel`, are the trace of the platform, not an explanation of it.

A snapshot is refreshed on purpose, never by accident. The intended versions are installed, the analysis is run, its output is compared with the previous run, and only then is `python -m pip freeze > requirements-lock.txt` written, its header updated and the file committed together with the output that was checked. The snapshot of Listing 4 holds `openai==3.31.0` while Listing 3 installed 3.35.1 six days later. The gap is the reason the snapshot exists.

```
$ python3.12 -m venv .venv
$ which python
/Users/boris/.local/bin/python
$ source .venv/bin/activate
(.venv) $ which python
/Users/boris/GitHub/attractor/.venv/bin/python
(.venv) $ echo "${PATH%%:*}"
/Users/boris/GitHub/attractor/.venv/bin
(.venv) $ deactivate
$ which python
/Users/boris/.local/bin/python
```

Listing 2: Creating and activating a virtual environment. Activation changes the first entry of `PATH`, hence the `python` that `which` finds; `deactivate` undoes it.

```
(.venv) $ python -m pip install -q -r requirements.txt
(.venv) $ python -m pip freeze | wc -l
      52
(.venv) $ python -m pip freeze | grep -E '^(numpy|matplotlib|python-dotenv|openai|
    ipykernel)=='
ipykernel==7.4.0
matplotlib==3.11.2
numpy==2.5.4
openai==3.35.1
python-dotenv==1.2.4
```

Listing 3: Installing the specification and reading back the snapshot. The versions are those available on 20 October 2026 and satisfy `numpy>=1.26`, which admits NumPy 2.

2 Package versions

2.1 Version specifiers

Definition 2.1: Version specifier

A **version specifier** is the clause after a package name in a requirement, such as `>=1.26`, and it selects the releases the installer may choose [6]. Clauses are combined with commas, `>=1.26,<2`, and all must hold. The operators are `==` (this release), `!=`, `<`, `<=`, `>`, `>=`, the compatible-release operator `~=`, and the wildcard `==1.26.*`. A version with fewer components than the release, as in `==1.26`, is padded with zeros: it names 1.26.0 and no other release.

Table 2 applies six clauses to four NumPy releases, two of the 1.x series and two of the 2.x series. The matches were computed with the `packaging` library, which pip uses, and the exercise of The first exercise (page 21) repeats the computation. The lower bound alone admits the 2.x releases, the upper bound `<2` excludes them, and `~=1.26.0` means "1.26.0 or later within 1.26", the clause to write when bug fixes are wanted and the major version must not change.

`pip install numpy==1.26` does not install "NumPy 1.26". The clause `==1.26` is padded to `==1.26.0` and matches release 1.26.0 only, not 1.26.4, the last of the series and the one in the snapshot. A series is named with the wildcard, `==1.26.*`, or the compatible-release operator, `~=1.26.0`.

On the command line a clause containing < or > is quoted, `"numpy>=1.26,<2"`, because the shell

```
# pip freeze of a Python 3.12.12 virtual environment (macOS, arm64) on 2026-10-14,
# after: pip install -r requirements.txt "numpy==1.26.4" "matplotlib<3.10"
# numpy 1.26.4 is the last 1.x release; it has wheels for Python 3.9 to 3.12 only.
annotated-types==0.8.0
anyio==4.15.1
appnope==1.0.0
asttokens==3.0.2
...                                    # 52 lines in all
matplotlib==3.9.4
numpy==1.26.4
openai==3.31.0
python-dotenv==1.2.4
...
```

Listing 4: The head of the committed snapshot `requirements-lock.txt`: a header written by hand, then 52 pinned distributions in alphabetical order as `pip freeze` prints them.

Table 2: Six version specifiers applied to four NumPy releases. A tick marks a release the installer may choose, and pip takes the newest of them.

clause	meaning	1.26.0	1.26.4	2.0.0	2.5.4
`>=1.26`	1.26.0 or later, any major version	✓	✓	✓	✓
`>=1.26,<2`	the 1.x series from 1.26.0 on	✓	✓		
`==1.26`	release 1.26.0 only, by zero padding	✓			
`==1.26.4`	release 1.26.4 only		✓		
`==1.26.*`	every release of the 1.26 series	✓	✓		
`~=1.26.0`	1.26.0 or later within 1.26	✓	✓		

reads an unquoted < as input redirection and an unquoted > as output redirection. The header of Listing 4 shows the quoted form.

2.2 Breaking releases and interpreter-bound wheels

A new major version of a library may remove names and change behaviour. NumPy 2.0, released in June 2024, removed the alias `np.NaN` in favour of `np.nan` and changed the promotion rules of scalars, among other things listed in its migration guide [7]. Listing 5 runs `analyse.py` in the environment of Listing 3, Python 3.12 with NumPy 2.5.4. The traceback ends in NumPy's own `__getattr__`, which raises the error with the replacement in the message. The same environment built with Python 3.13 fails identically, which was the demonstration of the lecture (Listing 6).

Two remedies exist, and they are not alternatives. The code is migrated, here by replacing `np.NaN` with `np.nan`, which also works with NumPy 1.26, and the snapshot is refreshed once the output has been checked. Until then the specification can exclude the breaking series with a tested upper bound, `numpy>=1.26,<2`. Listing 6 makes the migration on a copy with `sed`, so that the original file stays as committed, and runs the copy in a second environment built with Python 3.13. The table of periods is the one of the 2024 run.

The printed table agrees to three decimals. The files do not agree byte for byte (Listing 7). `load_run` reads the data as `float32`, and the period is a `float32` difference of two times divided by a Python `int`. NumPy 1 promotes that quotient to `float64` and writes 17 digits. NumPy 2 follows NEP 50 [8], keeps the `float32` and writes the shortest decimal that reads back to the same `float32`, 8.059375. Both strings denote the same binary value, the `float32` nearest to 8.0594 days, and both round to 8.059. A comparison of result files between two environments

```
(.venv) $ python analyse.py
Traceback (most recent call last):
  File "/Users/boris/GitHub/attractor/analyse.py", line 85, in <module>
    main()
  File "/Users/boris/GitHub/attractor/analyse.py", line 66, in main
    table = np.full((len(files), width), np.NaN)
                                         ^^^^^^
  File "/Users/boris/GitHub/attractor/.venv/lib/python3.12/site-packages/numpy/
    __init__.py", line 763, in __getattr__
    raise AttributeError(
AttributeError: ‘np.NaN‘ was removed in the NumPy 2.0 release. Use ‘np.nan‘ instead.
```

Listing 5: The removed alias. The traceback points at the line of `analyse.py` that uses `np.NaN` and at the NumPy 2.5.4 module that raises the error.

must therefore be made with a tolerance, as the previous chapter on numbers in memory explained, and not with `diff`.

The second environment also shows why the snapshot names its interpreter.

Definition 2.2: Wheel

A **wheel** is the built, ready-to-copy form of a Python distribution, a zip archive with the extension `.whl` [9]. A package with compiled extensions, such as NumPy, publishes one wheel per interpreter version and platform, named like `numpy-1.26.4-cp312-cp312-macosx_11_0_arm64.whl`: CPython 3.12, macOS 11 or later, arm64. pip installs a wheel when one matches the running interpreter and otherwise falls back to the **source distribution**, a `.tar.gz` archive it must compile on the spot.

NumPy 1.26.4 was released in February 2024, before Python 3.13 existed, and ships wheels for CPython 3.9 to 3.12 only [10]. With Python 3.13, the snapshot cannot be installed from wheels (Listing 8). The option `--only-binary=:all:` forbids the fallback, so pip stops with the list of the NumPy releases that have a 3.13 wheel, which starts at 2.1.0. Without the option, pip downloads `numpy-1.26.4.tar.gz`, 15.8 MB, and compiles it, which took 21 s on the lecture laptop with its C compiler and fails outright on a machine without one. The snapshot is bound to the interpreter series it was taken with, and its header says so. This is a fact about the releases of 2024 to 2026: later NumPy 1.x releases do not exist, and later CPython releases will never gain a 1.26.4 wheel.

3 Configuration outside the code

3.1 Configuration in environment variables

The third layer holds the values that change from machine to machine and from run to run without any change to the code: where the data are, where the results go, how many threads a library may use, which server a script talks to. The scripts read them from environment variables, which the previous chapter on the command line introduced with `export` and `printenv`.

Definition 3.1: Environment variable

An **environment variable** is a named string held by a process and copied to every child process it starts [11]. In the shell, `export NAME=value` sets a variable and marks it for this copy. A bare assignment `NAME=value` sets a shell variable that the shell itself can read and that child processes do not receive. The prefix form `NAME=value command` exports

```
$ python3.13 -m venv .venv313 && .venv313/bin/python -m pip install -q -r
    requirements.txt
$ .venv313/bin/python -m pip freeze | grep -E '^(numpy|matplotlib)=='
matplotlib==3.11.2
numpy==2.5.4
$ .venv313/bin/python analyse.py 2>&1 | tail -1
AttributeError: `np.NaN` was removed in the NumPy 2.0 release. Use `np.nan` instead.
$ sed 's/np\.NaN/np.nan/g' analyse.py > analyse_np2.py
$ RESULTS_PATH=results-np2 .venv313/bin/python analyse_np2.py
run    samples  peaks  period (days)
C1         359     56          6.600
C2         189     36          4.971
C3         182     30          5.614
C4         139     20          7.159
C5          74     12          6.671
C6         231     25          9.251
C7         234     25          7.002
C8         184     22          8.050
C9         141     17          8.059
C10        197     28          6.630
written: results-np2/periods.csv, results-np2/periods.png
```

Listing 6: A second environment with Python 3.13.11 and the specification, hence NumPy 2.5.4; the corrected copy prints the same table as the snapshot environment. The results go to a second directory, which `.gitignore` also covers.

```
$ grep C9 results/periods.csv results-np2/periods.csv
results/periods.csv:C9,141,17,8.059374809265137
results-np2/periods.csv:C9,141,17,8.059375
$ .venv/bin/python -c "import numpy as np; print(np.__version__, type(np.float32(1)
    / 2).__name__)"
1.26.4 float64
$ .venv313/bin/python -c "import numpy as np; print(np.__version__, type(np.float32
    (1) / 2).__name__)"
2.5.4 float32
```

Listing 7: The C9 line of `periods.csv` written by NumPy 1.26.4 and by NumPy 2.5.4, and the type of a `float32` scalar divided by an `int` under each.

```
$ python3.13 -m venv .venv313lock
$ .venv313lock/bin/python -m pip install --only-binary=:all: -r requirements-lock.
    txt
...
ERROR: Could not find a version that satisfies the requirement numpy==1.26.4 (from
    versions: 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.2.0, 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5,
    2.2.6, 2.3.0, 2.3.1, 2.3.2, 2.3.3, 2.3.4, 2.3.5, 2.4.0rc1, 2.4.1, 2.4.2, 2.4.3,
    2.4.4, 2.4.5, 2.4.6, 2.5.0rc1, 2.5.0, 2.5.1, 2.5.2, 2.5.3, 2.5.4)
ERROR: No matching distribution found for numpy==1.26.4
```

Listing 8: The snapshot on Python 3.13 with wheels only. pip finds no `numpy==1.26.4` wheel for this interpreter and lists the releases that have one, from 2.1.0 to 2.5.4 on 20 October 2026.

the variable to that one command only. In Python, `os.environ` is the dictionary of the variables the process received, and `os.environ.get(name, default)` returns the default when the name is absent [12].

Table 3 lists the variables the project reads, with the effect of each and the value used when it is unset. Listing 9 redirects one run with the prefix form. The script writes to `/tmp/attractor-out` and the files of the earlier run in `results/` are untouched. The same value written in `.env` would redirect every run.

Table 3: The environment variables read by the `attractor` scripts and by `setup_env.sh`.

variable	effect	value when unset
`DATA_PATH`	directory holding `C1.csv` to `C10.csv`, read by `analyse.py`	`data`
`RESULTS_PATH`	directory receiving `periods.csv` and `periods.png`	`results`
`OMP_NUM_THREADS`	thread limit for the BLAS library behind NumPy and other OpenMP code	the library's default
`MPLBACKEND`	Matplotlib backend, `Agg` writing image files without opening a window [13]	`Agg`, set by the script
`LLM_BASE_URL`	OpenAI-compatible chat endpoint used by `ask_llm.py`	none: the script stops
`LLM_API_KEY`	key sent to the endpoint, any string for a local `llama-server`	`EMPTY`
`LLM_MODEL`	model name requested from the endpoint	`qwen3.8-flash-next`
`PYTHON`	interpreter with which `setup_env.sh` creates `.venv`	`python3.12`

A bare assignment is not seen by the script. After `RESULTS_PATH=/tmp/attractor-out` the shell's `echo $RESULTS_PATH` prints the value, while `python` started from that shell finds nothing in `os.environ` and writes to `results/`. The value reaches the script through `export`, or through the prefix form of Listing 9.

A variable set to the empty string is set, not absent. With the line `DATA_PATH=` in `.env` or in the shell, `os.environ.get("DATA_PATH", "data")` returns `""`, the script searches the current directory for the runs and stops with `no C*.csv found in .` (Listing 9). The default applies only when the name is missing from the environment.

```
(.venv) $ RESULTS_PATH=/tmp/attractor-out python analyse.py | tail -1
written: /tmp/attractor-out/periods.csv, /tmp/attractor-out/periods.png
(.venv) $ RESULTS_PATH=/tmp/attractor-out
(.venv) $ echo $RESULTS_PATH
/tmp/attractor-out
(.venv) $ python -c "import os; print(os.environ.get('RESULTS_PATH'))"
None
(.venv) $ export RESULTS_PATH
(.venv) $ python -c "import os; print(os.environ.get('RESULTS_PATH'))"
/tmp/attractor-out
(.venv) $ DATA_PATH= python analyse.py
analyse.py: no C*.csv found in . (set DATA_PATH)
```

Listing 9: One run redirected with the prefix form, the difference between a bare assignment and an exported variable, and the effect of an empty value.

3.2 The .env file and python-dotenv

Typing exports at every prompt does not scale and leaves no record. The values live in a file of assignments, `NAME=value` without spaces, one per line, with `#` for comments. The repository commits a template, `.env.example`, with every variable, a default and a one-line comment (Listing 10). The working copy, `.env`, is created by copying the template and editing the values for the machine at hand, and it is listed in `.gitignore`. Secrets such as an API key and addresses of private services are written in `.env` only, so that they never reach a public repository.

```
$ grep '^[A-Z]' .env.example
DATA_PATH=data
RESULTS_PATH=results
OMP_NUM_THREADS=4
MPLBACKEND=Agg
LLM_BASE_URL=http://localhost:8080/v1
LLM_API_KEY=EMPTY
LLM_MODEL=qwen3.8-flash-next
$ cp .env.example .env
$ set -a; source .env; set +a
```

Listing 10: The assignments of the template, read with `grep '^[A-Z]'`, which prints the lines starting with a capital letter and skips the comment preceding each; then the three commands that load the local copy into the shell.

The shell loads the file with three commands. `source .env` runs the assignments in the current shell. `set -a`, the `allexport` option, marks every variable assigned from then on for export, so that a plain `NAME=value` line reaches child processes as if it carried `export` [14]. `set +a` switches the option off again. A value already set in the shell is overwritten by the file. The scripts apply the opposite rule.

The scripts do not depend on the shell having loaded the file. The package `python-dotenv` [15], version 1.2.4 in the snapshot, reads it from Python (Listing 11). `load_dotenv()` looks for `.env` from the directory of the script upwards and copies its assignments into `os.environ`. By default it does not override a variable that is already set, so a value exported by the shell or given in the prefix form takes precedence over the file. `os.environ.get` with a default makes every variable optional, and the scripts run from a bare clone without any `.env`. The `setdefault` line for `MPLBACKEND` runs before `import matplotlib.pyplot`, because the backend is chosen at import time.

```
from dotenv import load_dotenv

load_dotenv()                                        # reads .env, if present
DATA = Path(os.environ.get("DATA_PATH", "data"))     # default when unset
RESULTS = Path(os.environ.get("RESULTS_PATH", "results"))
os.environ.setdefault("MPLBACKEND", "Agg")           # before importing matplotlib
import matplotlib.pyplot as plt  # noqa: E402  (after the backend is set)
```

Listing 11: The configuration lines of `analyse.py`: the file is read, each variable gets a default, and the Matplotlib backend is fixed before the import that reads it.

`check_env.py` prints what both mechanisms produced: the interpreter, NumPy with the file it was imported from, `PYTHONPATH`, and every variable with its value or its default (Listing 12). The key is printed as `set` or `not set` only, so that the report can be pasted into a message without leaking it. On a fresh clone `LLM_BASE_URL` shows the placeholder of the

template. On the lecture laptop, whose `.env` holds the course endpoint, the same line reads `https://<course-endpoint>/v1`.

> Editing `.env` changes nothing in a shell that has already loaded it. The shell still holds the exported old value, and `load_dotenv()` does not override a set variable, so the scripts keep the old value too. After every edit of `.env`, `source setup_env.sh` is run again, or the variable is exported by hand.

```
(.venv) $ python check_env.py
python          /Users/boris/GitHub/attractor/.venv/bin/python
version         3.12.12
numpy           1.26.4
numpy file      /Users/boris/GitHub/attractor/.venv/lib/python3.12/site-packages/
    numpy/__init__.py
PYTHONPATH      (unset)
DATA_PATH       data
RESULTS_PATH    results
OMP_NUM_THREADS 4
MPLBACKEND      Agg
LLM_BASE_URL    http://localhost:8080/v1
LLM_MODEL       qwen3.8-flash-next
LLM_API_KEY     set
```

Listing 12: The environment report on a fresh clone, from the snapshot environment of Section 1.3. The last three values come from the template copied to `.env`.

3.3 A service address in the configuration

The script `ask_llm.py` sends the table of periods to a language model and prints the shortest and the longest period. The address of the model, the key and the model name are the three `LLM_` variables of Table 3, and they are the clearest case for the configuration layer: the committed template holds a placeholder, `http://localhost:8080/v1`, and the lecture laptop's `.env` holds the course endpoint, which is private. `LLM_BASE_URL` names any server that speaks the OpenAI chat-completions protocol: the course endpoint, a `llama-server` [16] or Ollama on the laptop itself, `http://localhost:11434/v1`, or `https://api.openai.com/v1` with a real key and model name.

Listing 13 shows how the script guards its configuration. A missing address stops the script before any network call. An address at which nothing listens raises `APIConnectionError` in the `openai` client [17], version 3.31.0 in the snapshot, which the script turns into a one-line message and exit status 1. The request itself sets `temperature=0` to reduce sampling variation and switches off the model's reasoning preamble through `enable_thinking`, a parameter of the llama.cpp server.

Listing 14 is retained from the lecture of 15 October 2026. The endpoint was a `llama-server` serving `qwen3.8-flash-next`, and the call took 1.6 s. The second session is a fresh clone with the placeholder still in `.env`: the client cannot connect to port 8080 of the laptop and the script stops as designed. Without any `.env`, the first guard stops it with `LLM_BASE_URL is not set`.

4 Rebuilding the environment

```
base_url = os.environ.get("LLM_BASE_URL")
if not base_url:
    raise SystemExit("ask_llm.py: LLM_BASE_URL is not set; copy .env.example to .env
     and fill it in")
model = os.environ.get("LLM_MODEL", "qwen3.8-flash-next")
client = OpenAI(base_url=base_url, api_key=os.environ.get("LLM_API_KEY", "EMPTY"))
try:
    reply = client.chat.completions.create(model=model, temperature=0, max_tokens
    =200,
        extra_body={"chat_template_kwargs": {"enable_thinking": False}}, messages
    =[...])
except APIConnectionError:
    raise SystemExit(f"ask_llm.py: cannot reach {base_url}; set LLM_BASE_URL in .env
     and run source setup_env.sh again")
```

Listing 13: The configuration guards of `ask_llm.py`. The two `SystemExit` messages are the ones printed in Listing 14.

```
(.venv) $ python ask_llm.py                          # after python analyse.py
Shortest: C2, 4.971 days
Longest: C6, 9.251 days
(.venv) $ python ask_llm.py                          # fresh clone, placeholder in .env
ask_llm.py: cannot reach http://localhost:8080/v1; set LLM_BASE_URL in .env and run
    source setup_env.sh again
(.venv) $ echo $?
1
```

Listing 14: `ask_llm.py` against the course endpoint, as recorded in the lecture, and on a fresh clone with the placeholder address.

4.1 The setup script

`setup_env.sh` turns the committed files into a working shell in one command, `source setup_env.sh`, from the repository root in bash or zsh. Its steps are:

1. refuse to continue when run instead of sourced (below), or when started from a directory without the snapshot;
2. if `.venv` is absent, create it with `python3.12`, or with the interpreter named in `PYTHON`;
3. `. .venv/bin/activate`;
4. if the snapshot is newer than the stamp file `.venv/lock.stamp`, the test `-nt`, install `requirements-lock.txt`, register the Jupyter kernel `attractor-py312`, and touch the stamp;
5. if `.env` is absent, copy `.env.example` to `.env`;
6. `set -a; . ./.env; set +a;`
7. `python check_env.py`.

A second `source setup_env.sh` runs only steps 3, 6 and 7, and the stamp makes the script a cheap command to repeat.

The first step rests on the difference between running and sourcing a script, which the previous chapter introduced and Figure 2 draws. `./setup_env.sh` starts a new bash process, a child of the interactive shell. The child receives a copy of the exported variables, and everything it changes, `PATH` after activation and the variables exported from `.env`, lives in that copy and vanishes when the child exits. `source setup_env.sh`, or `. setup_env.sh`, reads the file into

the current shell [18], so that activation changes the PATH of the shell at the prompt. The script detects which case it is in: in bash, BASH_SOURCE[0] equals $0 only when run, and in zsh, ZSH_EVAL_CONTEXT ends in :file only when sourced. Listing 15 shows the refusal and its exit status.



Figure 2: Running versus sourcing. A run script works in a child process, whose changes are lost at exit. A sourced script changes the shell at the prompt, which is the only way an activation can outlive the script.

A script cannot activate a virtual environment for the shell that started it. ./setup_env.sh, bash setup_env.sh and zsh setup_env.sh all run in a child process, and the PATH they set and the variables they export disappear when it exits. Only source setup_env.sh leaves the prompt shell activated.

setup_env.sh: run it from the repository root is printed by the script's first test, [-f requirements-lock.txt], and means that the snapshot is not in the current directory: the shell is in a subdirectory, or the copy of the project is incomplete. A fresh clone contains the file.

```
$ ./setup_env.sh; echo "exit: $?"
setup_env.sh: use 'source setup_env.sh'; a script that is run cannot activate the
    venv in the calling shell
exit: 1
$ bash setup_env.sh; echo "exit: $?"
setup_env.sh: use 'source setup_env.sh'; a script that is run cannot activate the
    venv in the calling shell
exit: 1
$ cd data && source ../setup_env.sh; echo "exit: $?"
setup_env.sh: run it from the repository root
exit: 1
```

Listing 15: The script refuses to be run, in three ways, and refuses to be sourced from a directory without the snapshot. All four stop with exit status 1.

Listing 16 is the first run on the lecture laptop after rm -rf .venv results, timed. time needs a child shell to time a builtin such as source, so the timed command is bash -c 'source setup_env.sh': the child's activation is lost, exactly as Figure 2 says, and the working command at the prompt remains source setup_env.sh. The first run took 12.4 s with a warm pip cache, the second 0.075 s, since steps 2, 4 and 5 are skipped. A laptop that has never downloaded these wheels fetches about 50 MB and typically takes tens of seconds. The kernel line names the interpreter the notebook kernel will use (Section 4.3).

Listing 17 is retained from the lecture: the same script on a Linux workstation reached over ssh, where the project had been copied without .venv, results, .env and .git. The system interpreter is Python 3.12.3 rather than the laptop's 3.12.12. The snapshot pins packages, not

```
$ rm -rf .venv results
$ time bash -c 'source setup_env.sh'
setup_env.sh: creating .venv with Python 3.12.12 at /opt/homebrew/bin/python3.12
setup_env.sh: installing requirements-lock.txt
setup_env.sh: Jupyter kernel 'attractor-py312' -> /Users/boris/GitHub/attractor/.
    venv/bin/python
setup_env.sh: .env created from .env.example; edit it to set LLM_BASE_URL
python          /Users/boris/GitHub/attractor/.venv/bin/python
version         3.12.12
numpy           1.26.4
...
LLM_API_KEY     set
bash -c 'source setup_env.sh'  7.19s user 2.43s system 77% cpu 12.353 total
$ time zsh -c 'source setup_env.sh'       # second run: the report only
...
zsh -c 'source setup_env.sh'  0.10s user 0.09s system 255% cpu 0.075 total
$ source setup_env.sh                     # the working command: activates this
    shell
```

Listing 16: First and second run of the setup script on the lecture laptop, timed in a child shell. The first run creates the environment, installs the snapshot, registers the kernel and creates `.env`; the second only reports.

the patch level of the interpreter, and every pinned release has a `manylinux` wheel for CPython 3.12, so the install takes the same 12 s and the analysis prints the same table. The kernel goes to the Linux location, `~/.local/share/jupyter/kernels/attractor-py312/`, and `.env` holds the placeholder address until it is edited.

4.2 The import search path

Definition 4.1: Import search path

`sys.path` is the ordered list of directories in which `import` looks for a module, and the first directory holding a module of the requested name wins [19]. For a script its first entry is the directory of the script. Then come the directories of the environment variable `PYTHONPATH`, if it is set, then the standard library, and last the `site-packages` directory of the active environment, where pip installs [20]. The attribute `__file__` of an imported module names the file that was chosen.

The order is the trap (Figure 3). A `PYTHONPATH` exported in `~/.zshrc` reaches every Python process started from that shell, including the interpreter of an activated virtual environment. If one of its directories holds a package with the same name as an installed one, that copy is imported instead of the one in the environment, and `pip freeze` gives no sign of it, since pip lists what it installed and not what `import` selected. Listing 18 builds the situation with a stub: a directory `old_code` holding a `numpy` package of one line. With the variable set, `import numpy` returns the stub, and `analyse.py` fails on the first name the stub lacks. The report of Listing 12 prints both `PYTHONPATH` and the file NumPy was imported from for this reason, and the file must point into the environment's `site-packages`.

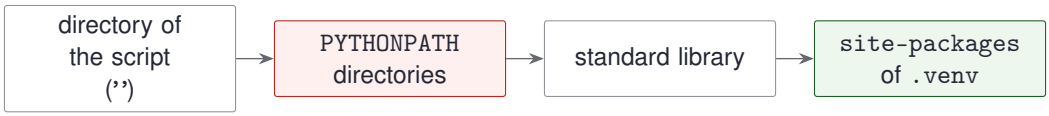
`PYTHONPATH` does not replace the search path, it is inserted at the front of it. The environment's `site-packages` is still searched, but only for names that no earlier directory provides, so a stale copy of a library on `PYTHONPATH` shadows the installed one. A file next to the script named like a library, `numpy.py` or `random.py`, shadows it in the same way through the first entry.

```
$ ssh orion
$ cd ~/attractor-l4test
$ which python3.12; python3.12 --version
/usr/bin/python3.12
Python 3.12.3
$ time bash -c 'source setup_env.sh'
setup_env.sh: creating .venv with Python 3.12.3 at /usr/bin/python3.12
setup_env.sh: installing requirements-lock.txt
setup_env.sh: Jupyter kernel 'attractor-py312' -> /home/cmbagent/attractor-l4test/.
    venv/bin/python
setup_env.sh: .env created from .env.example; edit it to set LLM_BASE_URL
python          /home/cmbagent/attractor-l4test/.venv/bin/python
version         3.12.3
numpy           1.26.4
...
LLM_BASE_URL    http://localhost:8080/v1
LLM_MODEL       qwen3.8-flash-next
LLM_API_KEY     set

real    0m11.899s
user    0m10.458s
sys     0m0.793s
$ source setup_env.sh >/dev/null && python analyse.py && python period.py data/C9.
    csv
run    samples  peaks  period (days)
C1         359     56          6.600
...
C10        197     28          6.630
written: results/periods.csv, results/periods.png
data/C9.csv: 141 samples, 17 peaks, period 8.059 days
```

Listing 17: The setup script on the workstation, Ubuntu 24.04 with its system Python 3.12.3, as recorded in the lecture of 15 October 2026. The table of periods is the one of Listing 6.

`unset PYTHONPATH` removes the variable from the current shell. The permanent fix is to delete the `export PYTHONPATH=...` line from `~/.zshrc` and open a new terminal. The legitimate uses of the variable are short-lived: a one-off import test, `PYTHONPATH=$PWD/src python script.py`, for one command. A project with packaging metadata is made importable with `python -m pip install -e .` instead, which registers it in the active environment and nowhere else, the subject of the chapter on packaging.



`sys.path`, searched left to right. The first directory holding the name wins.

Figure 3: The import search path of a script run in a virtual environment. A `PYTHONPATH` directory, or a file next to the script, is found before the packages installed in the environment.

```
(.venv) $ mkdir -p old_code/numpy && echo '__version__ = "1.21.0"' > old_code/numpy/
    __init__.py
(.venv) $ python -c "import numpy; print(numpy.__version__, numpy.__file__)"
1.26.4 /Users/boris/GitHub/attractor/.venv/lib/python3.12/site-packages/numpy/
    __init__.py
(.venv) $ export PYTHONPATH=$PWD/old_code
(.venv) $ python -c "import numpy; print(numpy.__version__, numpy.__file__)"
1.21.0 /Users/boris/GitHub/attractor/old_code/numpy/__init__.py
(.venv) $ python -c "import sys; print(sys.path[:2])"
['', '/Users/boris/GitHub/attractor/old_code']
(.venv) $ python analyse.py 2>&1 | tail -1
ImportError: cannot import name 'VisibleDeprecationWarning' from 'numpy' (/Users/
    boris/GitHub/attractor/old_code/numpy/__init__.py)
(.venv) $ unset PYTHONPATH
```

Listing 18: A stale copy on `PYTHONPATH` shadows the installed NumPy. The stub is one line; the variable puts its directory second in `sys.path`, before the standard library and `site-packages`.

4.3 Jupyter kernels

Definition 4.2: Jupyter kernel

A **Jupyter kernel** is the process that executes the cells of a notebook [21]. Its **kernelspec**, a file `kernel.json` in a kernels directory of the user, records the command that starts it, beginning with the path of a Python executable. The notebook front end, JupyterLab or VS Code, lists every kernelspec it finds under the display name recorded in the file. `python -m ipykernel install --user --name NAME` writes the kernelspec for the interpreter that runs the command [22].

The setup script registers the kernel `attractor-py312`, display name `Python (attractor 3.12)`, each time it installs the snapshot, so that the kernelspec names the current `.venv/bin/python` (Listing 19). The file lives in a directory named after the kernel, under `~/Library/Jupyter/kernels/` on the laptop and under `~/.local/share/jupyter/kernels/` on Linux. The first cell of a notebook checks the kernel: `import sys; sys.executable` must end in `attractor/.venv/bin/python`. A kernel naming another interpreter, the Homebrew `python3.12` itself for instance, reports `ModuleNotFoundError` for everything installed in `.venv`, while the same script runs in the terminal. A virtual environment deleted and recreated at the same path keeps the kernelspec valid, and one moved elsewhere breaks it until `ipykernel install` is run again.

A kernel registered on the laptop is visible to JupyterLab and VS Code on the laptop. A hosted Google Colab runtime runs on a machine of Google's and does not see it. Colab reaches a local kernel only through "Connect to a local runtime", which requires a Jupyter server running on the laptop [23].

4.4 uv, conda and Colab

The course builds environments with `venv` and installs packages with pip. Three other tools come up in research groups.

`uv` [24] is a package manager that replaces the `venv` and pip commands with faster equivalents of the same meaning: `uv venv` creates an environment, `uv pip install -r requirements.txt` installs into it, and `uv python install 3.12` downloads an interpreter of that series, which is the simplest way to obtain Python 3.12 on a machine that lacks it. The project's `setup_env.sh`

```
(.venv) $ jupyter kernelspec list | grep attractor
  attractor               /Users/boris/Library/Jupyter/kernels/attractor
  attractor-py312         /Users/boris/Library/Jupyter/kernels/attractor-py312
(.venv) $ head -4 ~/Library/Jupyter/kernels/attractor-py312/kernel.json
{
 "argv": [
  "/Users/boris/GitHub/attractor/.venv/bin/python",
  "-Xfrozen_modules=off",
```

Listing 19: The kernelspec written by the setup script. `jupyter kernelspec list` prints every kernel the front ends will offer; the `argv` of `kernel.json` starts with the interpreter of the environment.

then takes it through `PYTHON="$(uv python find 3.12)"`. The files of Table 1 are unchanged.

conda [25] manages an environment that can include the interpreter itself, Python packages and compiled libraries that are not Python at all, such as CUDA, MPI or HDF5. Its packages come from conda channels, `conda-forge` being the community one, and its `environment.yml` plays the role of the specification. `conda env export` records the resolved environment with platform-specific builds, the role of the snapshot, and `conda env export --from-history` records only what was requested. mamba is the same command interface with a faster solver, and Miniforge [26] installs both with `conda-forge` preset. Installing with pip into a conda environment works, but conda's record of the environment then no longer describes it.

```
name: attractor
channels: [conda-forge]
dependencies:
  - python=3.12
  - numpy
  - matplotlib
```

Listing 20: A conda specification equivalent to `requirements.txt`, with the interpreter as one of the dependencies.

A hosted Google Colab runtime supplies its own interpreter and kernel, on a machine of Google's, with a set of preinstalled packages. The project is cloned into the runtime and its specification installed into the running kernel (Listing 21). No `.venv` is created, the files disappear when the runtime ends, and Google Drive is mounted for files that must persist. The available runtime images, as of October 2026, run Python 3.12 or 3.11 [27], and a runtime with NumPy 2 needs the migration of Section 2.2.

In a notebook cell, `!` and `%` are different mechanisms [28]. A line starting with `!` is passed to the system shell, so `!git clone` runs git. A line starting with `%` is an IPython magic: `%cd` changes the working directory of the kernel process, which `!cd` cannot do, and `%pip install` installs into the kernel's own interpreter, which `!pip` may not.

The Python version of a hosted Colab runtime is chosen, not imposed: Runtime, Change runtime type, Runtime version selects one of the available images, each with its bundled Python and packages. The selector installs no arbitrary interpreter, and no image provides Python 3.13 or 3.14 as of October 2026.

```
!git clone https://github.com/borisbolliet/attractor.git
%cd attractor
import sys
print(sys.version)
%pip install -r requirements.txt
from google.colab import drive
drive.mount('/content/drive')      # files that must outlive the runtime
```

Listing 21: Colab cells that clone the project, report the interpreter of the hosted runtime and install the specification into it.

Troubleshooting

Eight messages point back to the files and rules of this chapter.

symptom	cause	remedy
`error: externally-managed-environment` from `pip install`.	The interpreter is the operating system's or Homebrew's, which PEP 668 [29] marks as managed by its own package manager: no environment was created or activated (Section 1.2).	`which python` must print a path inside `.venv/bin`. Create and activate the environment. Neither `--break-system-packages` nor `sudo pip` is a fix.
`setup_env.sh: python3.12 not found.`	The snapshot needs Python 3.12 for its NumPy 1.26.4 wheel (Definition 2.2).	Install Python 3.12, with Homebrew, apt or `uv python install 3.12`, then `PYTHON=/path/to/python3.12 source setup_env.sh`.
`ERROR: No matching distribution found for numpy==1.26.4.`	No wheel of that release exists for the running interpreter, Python 3.13 here (Listing 8).	`rm -rf .venv`, then source the script with the interpreter of the snapshot. The script reuses an existing `.venv` whatever its interpreter.
`AttributeError: 'np.NaN' was removed in the NumPy 2.0 release.`	NumPy 2 where the code expects 1.x, after installing the specification instead of the snapshot (Section 2.2).	Install the snapshot, or migrate the code to `np.nan` and refresh the snapshot.
`ModuleNotFoundError: No module named 'dotenv'` in a notebook while `python check_env.py` works in the terminal.	The notebook runs another kernel (Definition 4.2).	`sys.executable` in the first cell names it. Select `Python (attractor 3.12)`, or register it again from the activated environment.
`ImportError` or wrong version from a package that `pip freeze` lists correctly.	A `PYTHONPATH` directory or a file next to the script shadows the installed package (Definition 4.1).	`module.__file__` shows the file imported. `unset PYTHONPATH`, and remove the export from `~/.zshrc`.
`setup_env.sh: use 'source setup_env.sh', exit status 1.`	The script was run, not sourced, so it cannot activate the environment for the prompt shell (Figure 2).	`source setup_env.sh` from the repository root.
`ask_llm.py: cannot reach http://localhost:8080/v1.`	`LLM_BASE_URL` holds the placeholder, or the server named is not running. An edited `.env` is not reloaded by a shell that exported the old value (Section 3.2).	`python check_env.py | grep LLM_BASE_URL` shows the value in force. Edit `.env`, then `source setup_env.sh` again.

Key points

- An environment is the interpreter, the packages and the configuration, each recorded in a committed file and rebuilt on every machine from it (Definition 1.1, Table 1).
- The specification says what the code needs and admits new major versions, which is

how `numpy>=1.26` installed NumPy 2.5.4 on any interpreter. The snapshot says what was run, `numpy==1.26.4`, and needs a wheel for the interpreter it was taken with (Definitions 1.2 and 2.2).

- `==1.26` names release 1.26.0 only. A series is `==1.26.*` or `~=1.26.0`, and `<2` keeps a major version out (Table 2).
- A variable reaches a script only when exported, by `export`, by the prefix form, or by `set -a` around `source .env`. A shell value takes precedence over `.env` in the scripts, and an empty value is not an absent one (Definition 3.1).
- A run script works in a child process and its activation dies with it, so the setup script must be sourced (Figure 2). `PYTHONPATH` is searched before the environment's packages, and a notebook runs whatever interpreter its kernelspec names (Definitions 4.1 and 4.2).

Course environment

The clone goes to `~/GitHub/attractor`: the first lectures filled `~/attractor`, and `git clone` refuses a destination that exists and is not empty. The kernel `attractor-py312` of this chapter is distinct from the kernel `attractor` registered in the first lecture for the Python 3.13 environment of `~/attractor`, and a notebook on the chemostat data selects the former.

On CSD3, `module load python/3.11.0-icl` supplies an interpreter [30], a virtual environment is created from it with `python -m venv`, and MPI and BLAS come from further modules rather than from pip. A project's `DATA_PATH` and `RESULTS_PATH` are the two values that change most between the laptop and the cluster, and they are exactly what `.env` is for.

On a cluster the data directory is a question of disk quota, not of memory. On CSD3, `/home` holds 50 GB per user and is meant for code and post-processed results, while `/rds/user/<username>/hpc-work` holds 1 TB and is where the data read and written by jobs belong [31]. `DATA_PATH` and `RESULTS_PATH` in the cluster's `.env` point there.

Exercises

Solutions are given from page 22.

Reading version specifiers

NumPy has published, among others, the releases 1.26.0, 1.26.4, 2.0.0 and 2.5.4. (a) For each of `numpy>=1.26`, `numpy==1.26`, `numpy~=1.26.0` and `numpy>=1.26,<2`, which of the four releases may pip install, and which one does it choose? (b) Write the clause for "any bug-fix release of the 1.26 series, and nothing else" in two ways. (c) The snapshot of Listing 4 pins `numpy==1.26.4`. Why is that clause right in the snapshot and wrong in the specification?

Where a variable comes from

The file `.env` of a clone contains `RESULTS_PATH=results`, and `analyse.py` reads the variable as in Listing 11. What does `python check_env.py` print for `RESULTS_PATH` in each situation, and why? (a) A new shell, nothing exported. (b) The prefix form, `RESULTS_PATH=/tmp/attractor-out` written before the command. (c) After `export RESULTS_PATH=/tmp/attractor-out`, the line in `.env` is edited to `RESULTS_PATH=results-b` and the report is run again. (d) After `unset RESULTS_PATH` in the same shell.

A stale copy on the import path

A colleague's `~/.zshrc` ends with `export PYTHONPATH=$HOME/old_code`, and `~/old_code` holds a NumPy checkout from 2021. (a) After `source setup_env.sh`, which NumPy does `python analyse.py` import, and which command shows it? (b) `pip freeze` prints `numpy==1.26.4`. Why does it not reveal the problem? (c) Give the one-command fix and the permanent fix, and state what each changes.

Solutions to the exercises

Reading version specifiers

(a) `>=1.26` admits all four and pip takes the newest, 2.5.4. `==1.26` is padded to 1.26.0 and admits that release only. `~=1.26.0` means `>=1.26.0,==1.26.*`, admits 1.26.0 and 1.26.4, and pip takes 1.26.4. `>=1.26,<2` admits the same two and also gives 1.26.4. (b) `numpy==1.26.*` or `numpy~=1.26.0`. (c) The snapshot records one environment that was run, so an exact pin is its purpose. The specification states what the code needs, and `==1.26.4` would forbid the bug-fix releases of 1.26 and any future series, including those the code would run with after a one-line migration. A lower bound, with an upper bound while the code is not migrated, says what is known.

Answer check

```
>>> from packaging.specifiers import SpecifierSet
>>> from packaging.version import Version
>>> rel = ["1.26.0", "1.26.4", "2.0.0", "2.5.4"]
>>> for spec in (">=1.26", "==1.26", "~=1.26.0", ">=1.26,<2", "==1.26.*"):
...     print(f"{spec:<10}", [r for r in rel if Version(r) in SpecifierSet(spec)
    ])
>=1.26     ['1.26.0', '1.26.4', '2.0.0', '2.5.4']
==1.26     ['1.26.0']
~=1.26.0   ['1.26.0', '1.26.4']
>=1.26,<2  ['1.26.0', '1.26.4']
==1.26.*   ['1.26.0', '1.26.4']
```

`packaging` 26.3 is in the snapshot, since pip depends on it.

Where a variable comes from

(a) `results`: nothing is exported, `load_dotenv()` copies the line of `.env` into `os.environ`. (b) `/tmp/attractor-out`: the prefix form exports the variable to the `python` process, and `load_dotenv()` does not override a variable that is set. (c) `/tmp/attractor-out` still: the shell exports the old value to every command it starts, and the edited file loses to it for the same reason. (d) `results-b`: the variable is gone from the shell, so the file is read again and its new line applies. The chain of precedence is the shell's exported value, then `.env`, then the default in the code.

Answer check

```
(.venv) $ python check_env.py | grep RESULTS_PATH
RESULTS_PATH    results
(.venv) $ RESULTS_PATH=/tmp/attractor-out python check_env.py | grep
    RESULTS_PATH
RESULTS_PATH    /tmp/attractor-out
(.venv) $ export RESULTS_PATH=/tmp/attractor-out
(.venv) $ sed -i.bak 's|^RESULTS_PATH=.*|RESULTS_PATH=results-b|' .env
(.venv) $ python check_env.py | grep RESULTS_PATH
RESULTS_PATH    /tmp/attractor-out
(.venv) $ unset RESULTS_PATH
(.venv) $ python check_env.py | grep RESULTS_PATH
RESULTS_PATH    results-b
```

A stale copy on the import path

(a) The 2021 checkout. `PYTHONPATH` directories come second in `sys.path`, before the environment's `site-packages`, so `import numpy` stops at `~/old_code/numpy`. `python -c "import numpy; print(numpy.__file__)"` shows the file, and `check_env.py` prints the same line as `numpy file`. (b) `pip freeze` lists the distributions installed in the environment, and NumPy 1.26.4 is installed there. It does not run `import` and cannot know which directory wins. (c) `unset PYTHONPATH` removes the variable from the current shell and its future children, for this terminal only. Deleting the `export` line from `~/.zshrc` and opening a new terminal removes it from every future shell. Neither touches the environment, which was correct all along.

Answer check

```
(.venv) $ PYTHONPATH=$PWD/old_code python -c "import numpy; print(numpy.__file__
    )"
/Users/boris/GitHub/attractor/old_code/numpy/__init__.py
(.venv) $ PYTHONPATH=$PWD/old_code python -m pip freeze | grep numpy
numpy==1.26.4
(.venv) $ python -c "import numpy; print(numpy.__file__)"
/Users/boris/GitHub/attractor/.venv/lib/python3.12/site-packages/numpy/__init__.
    py
```

The stub of Listing 18 stands in for the 2021 checkout. `pip freeze` is unchanged by the variable, and `import` is not.

References

[1] Boris Bolliet. attractor: cycle periods of ten chemostat runs. GitHub, https://github.com/borisbolliet/attractor, 2026. MIT licence; accessed 20 October 2026.

[2] Bernd Blasius, Lars Rudolf, Guntram Weithoff, Ursula Gaedke, and Gregor F. Fussmann. Long-term cyclic persistence in an experimental predator–prey system. *Nature*, 577:226–230, 2020. Data: https://doi.org/10.6084/m9.figshare.10045976, CC BY 4.0.

[3] Python Software Foundation. `venv`: creation of virtual environments. Python 3.12 documentation, https://docs.python.org/3/library/venv.html. accessed 20 October 2026.

[4] Free Software Foundation. Command search and execution. Bash reference manual, https://www.gnu.org/software/bash/manual/html_node/Command-Search-and-Execution.html. accessed 20 October 2026.

[5] pip developers. `pip freeze`. pip documentation, https://pip.pypa.io/en/stable/cli/pip_freeze/. accessed 20 October 2026.

[6] Python Packaging Authority. Version specifiers. Python Packaging User Guide, specifications, https://packaging.python.org/en/latest/specifications/version-specifiers/. The successor of PEP 440; accessed 20 October 2026.

[7] NumPy developers. NumPy 2.0 migration guide. NumPy 2.5 documentation, https://numpy.org/doc/stable/numpy_2_0_migration_guide.html. accessed 20 October 2026.

[8] Ralf Gommers and Sebastian Berg. NEP 50: Promotion rules for Python scalars. NumPy Enhancement Proposals, https://numpy.org/neps/nep-0050-scalar-promotion.html, 2021. Implemented in NumPy 2.0; accessed 20 October 2026.

[9] Python Packaging Authority. Binary distribution format. Python Packaging User Guide, specifications, https://packaging.python.org/en/latest/specifications/binary-distribution-format/. The wheel format, first defined in PEP 427; accessed 20 October 2026.

[10] NumPy developers. numpy 1.26.4. PyPI, https://pypi.org/project/numpy/1.26.4/, 2024. Release files: wheels for CPython 3.9 to 3.12 and a source archive; accessed 20 October 2026.

[11] Free Software Foundation. Environment. Bash reference manual, https://www.gnu.org/software/bash/manual/html_node/Environment.html. accessed 20 October 2026.

[12] Python Software Foundation. `os.environ`. Python 3.12 documentation, https://docs.python.org/3/library/os.html#os.environ. accessed 20 October 2026.

[13] Matplotlib developers. Backends. Matplotlib documentation, https://matplotlib.org/stable/users/explain/figure/backends.html. accessed 20 October 2026.

[14] Free Software Foundation. The `set` builtin. Bash reference manual, https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html. accessed 20 October 2026.

[15] Saurabh Kumar et al. python-dotenv. PyPI, https://pypi.org/project/python-dotenv/. Version 1.2.4 in the snapshot; accessed 20 October 2026.

[16] Georgi Gerganov et al. llama.cpp HTTP server. GitHub, https://github.com/ggml-org/llama.cpp/tree/master/tools/server. accessed 20 October 2026.

[17] OpenAI. openai-python: the official Python library for the OpenAI API. GitHub, https://github.com/openai/openai-python. accessed 20 October 2026.

[18] Free Software Foundation. Bash builtin commands: `source`. Bash reference manual, https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html. accessed 20 October 2026.

[19] Python Software Foundation. `sys.path`, `sys.executable`. Python 3.12 documentation, https://docs.python.org/3/library/sys.html#sys.path. accessed 20 October 2026.

[20] Python Software Foundation. Command line and environment: `PYTHONPATH`. Python 3.12 documentation, https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH. accessed 20 October 2026.

[21] Project Jupyter. Kernels. Jupyter documentation, https://docs.jupyter.org/en/latest/projects/kernels.html. accessed 20 October 2026.

[22] Project Jupyter. IPython kernel documentation. https://ipykernel.readthedocs.io/en/stable/. accessed 20 October 2026.

[23] Google. Colab local runtimes. https://research.google.com/colaboratory/local-runtimes.html. accessed 20 October 2026.

[24] Astral. uv: an extremely fast Python package and project manager. https://docs.astral.sh/uv/. Version 0.11 on the lecture laptop; accessed 20 October 2026.

[25] conda contributors. Managing environments. conda documentation, https://docs.conda.io/projects/conda/en/stable/user-guide/tasks/manage-environments.html. accessed 20 October 2026.

[26] conda-forge community. Miniforge. https://conda-forge.org/download/. accessed 20 October 2026.

[27] Google. Colab runtime versions. https://research.google.com/colaboratory/runtime-version-faq.html. accessed 20 October 2026.

[28] IPython developers. System shell access and magic commands. IPython documentation, https://ipython.readthedocs.io/en/stable/interactive/reference.html#system-shell-access. accessed 20 October 2026.

[29] Brett Cannon et al. PEP 668: Marking Python base environments as "externally managed". https://peps.python.org/pep-0668/, 2021. accessed 20 October 2026.

[30] University of Cambridge Research Computing Services. Modules. CSD3 user guide, https://docs.hpc.cam.ac.uk/hpc/user-guide/modules.html. accessed 20 October 2026.

[31] University of Cambridge Research Computing Services. Filesystems and I/O management. CSD3 user guide, https://docs.hpc.cam.ac.uk/hpc/user-guide/io_management.html. accessed 20 October 2026.

Index

configuration, 2

environment, 2
environment variable, 7

interpreter, 2

Jupyter kernel, 16

kernelspec, 16

lock file, 3

packages, 2

reproducible, 2

snapshot, 3
source distribution, 7
specification, 3

version specifier, 5
virtual environment, 2

wheel, 7