# Lecture 1 — Course environment

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

## Contents


**1 Python interpreters** **2**
- 1.1 Research computing and coding assistants . . . 2
- 1.2 The running example . . . 2
- 1.3 The Python interpreter . . . 3
- 1.4 Interpreters on disk and Python versions . . . 5

**2 Virtual environments and packages** **6**
- 2.1 Virtual environments . . . 6
- 2.2 Activation, pip and requirements . . . 7

**3 Notebooks and editors** **8**
- 3.1 Notebooks and kernels . . . 9
- 3.2 JupyterLab and VS Code . . . 9

**4 Remote machines** **11**
- 4.1 Connecting to a remote machine over SSH . . . 11
- 4.2 Python environments on a remote machine . . . 12
- 4.3 VS Code Remote-SSH . . . 12

**Troubleshooting** **13**

**Key points** **14**

**Course environment** **15**

**Exercises** **17**

**Solutions to the exercises** **18**


A computed result depends on the operating system, the shell, the Python interpreter and the installed libraries. A change at any of these levels can change the result without an error being raised, and a notebook that ran on a laptop in October may not run on a cluster in November. The running example of the course is the folder `attractor`, which holds the notebook `predator_prey_cycles.ipynb` and the data file `C9.csv`: 141 daily samples of a predator–prey experiment in which rotifers feed on green algae in a chemostat [1]. This chapter builds the environment in which that notebook runs, and runs it three times with the same output: in JupyterLab on the laptop, in VS Code on the laptop, and on a remote Linux machine reached over SSH.

The laptop outputs were produced on the lecture laptop with macOS 15.6 and Python 3.13.11 from Homebrew. The packages installed into the project environment were NumPy 2.5.4, Matplotlib 3.11.2 and ipykernel 7.4.0, and JupyterLab 4.6.4 was used as the front-end. In the listings the home directory is written `~`, and the shell's `PATH` was reduced to the four system directories shown in Listing 7. The remote machine is CSD3, the University of Cambridge cluster, whose outputs are those of October 2026 and whose Python module was 3.11.0. The course logistics, from the website to the CSD3 account, are collected at the end of the chapter (page 15).

# 1 Python interpreters

## 1.1 Research computing and coding assistants

Research code is scripted, versioned and packaged so that the whole stack that produced a result is recorded and can be rebuilt. Coding assistants now draft much of that code without seeing the machine it runs on. The person who reviews the draft needs a set of skills that the assistant cannot supply:

- predicting the effect of a shell command before running it,
- pinning the interpreter and library versions in use,
- reading a `git diff` and a commit history,
- writing a test that fails on wrong code,
- estimating the cost of a computation beforehand.

Each of these is the subject of a later chapter. This one starts with the first two: which interpreter runs a command, and which packages it sees. The course policy on assistants is stated at the end of the chapter (page 15).

## 1.2 The running example

The course grows one Python package, `attractor`, around the basics of dynamical systems, that is, systems evolving in time under a differential equation. It starts as a folder of four files, from which every command of this chapter runs:

- `predator_prey_cycles.ipynb`, the notebook of the demonstration,
- `C9.csv`, its data, which the notebook downloads from figshare when the file is absent,
- `requirements.txt`, the package list, three lines: `numpy>=1.26`, `matplotlib>=3.8` and `ipykernel>=6.29`,
- `period.py`, a five-line script used to show bytecode.

Listing 1 creates the folder from the archive on the lecture page [2]. The option `-j` of `unzip` drops the folder stored inside the archive. The same notebook opens in Google Colab from the lecture page, with no local installation.

```
$ mkdir ~/attractor && cd ~/attractor
$ curl -LO https://teaching.borisbolliet.io/c1-2026/materials/class01-demo.zip
$ unzip -j class01-demo.zip && rm class01-demo.zip
Archive:  class01-demo.zip
  inflating: predator_prey_cycles.ipynb
 extracting: requirements.txt
  inflating: period.py
  inflating: C9.csv
$ ls
C9.csv  period.py  predator_prey_cycles.ipynb  requirements.txt
```

**Listing 1:** The project folder, created from the archive of the lecture page. The archive holds a folder `class01-demo/` with the four files, which `unzip -j` unpacks without the folder.

The data come from a long-term chemostat experiment [1, 3]: a culture vessel diluted continuously with fresh medium, in which rotifers, the predator, feed on green algae, the prey. The file has one row per sample and seven columns: the time in days, the algae density in $10^6$ cells per ml, the rotifer density in animals per ml, and four further columns. Its first row is day 0.00,

with 1.92 for the algae and 16.25 for the rotifers. The run shown, C9, has a pulsed nutrient supply: the medium alternates between 160 and 0 µmol N/l every four days, so the dominant period of about 8 days in the series, Figure 1, is the period of the forcing. The notebook is only a vehicle for the tools of this chapter. The cycles call for a model, and building, testing and scaling that model is the work of the following chapters.

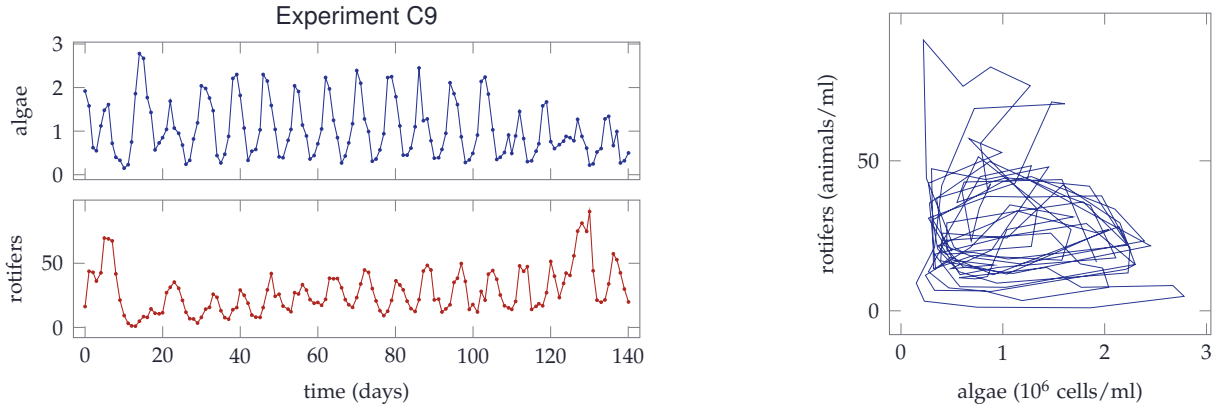



**Figure 1:** The data of `C9.csv` as the notebook draws them. Left: the algae density, in $10^6$ cells per ml, and the rotifer density, in animals per ml, against time. Right: the same series in the phase plane, where each predator–prey cycle traces one loop.

## 1.3 The Python interpreter

**Definition 1.1:** Interpreter and bytecode

The **Python interpreter** is a program that reads Python source and executes it. The reference interpreter, **CPython**, is written in C [4]. It first compiles the source to **bytecode**, instructions for its own evaluation loop [5], and then executes the bytecode one instruction at a time in that loop. Three kinds of code are available to a running interpreter: the **built-ins** such as `print` and `len`, always loaded; the **standard library**, the modules shipped with the interpreter [6]; and **third-party packages**, installed by `pip` into a directory named `site-packages`.

Listing 2 asks the interpreter for its standard library: 290 module names with Python 3.13.11, of which the names starting with an underscore are private, and the directory where the modules live. The count and the path belong to one interpreter. PyPy [7] is another implementation of the language, with a just-in-time compiler. The course uses CPython.

```
$ python3.13 -c "import sys; print(len(sys.stdlib_module_names))"
290
$ python3.13 -c "import sys; print(sorted(sys.stdlib_module_names)[:6])"
['__future__', '_abc', '_aix_support', '_android_support', '_apple_support', '_ast']
$ python3.13 -c "import sysconfig; print(sysconfig.get_paths()['stdlib'])"
/opt/homebrew/opt/python@3.13/Frameworks/Python.framework/Versions/3.13/lib/python3
    .13
```

**Listing 2:** The standard library of one interpreter, Python 3.13.11 from Homebrew: the number of modules, the first names in alphabetical order, and the directory that holds them.

The module `dis` [8] prints the bytecode of a script (Listing 3). The five-line script `period.py` defines a function and prints its value at 0.128, the frequency in cycles per day of the 7.8-day period found by the notebook. The main code object loads the constant 0.128, calls `period`

and prints the result. The code object of `period` loads the constant 1 and the argument `freq`, divides, and returns. The instruction names change between versions of the interpreter.

> Bytecode is not machine language. It is a set of instructions for CPython's evaluation loop, a program written in C, so the same bytecode runs unchanged on any processor. Machine language is what a C compiler produces for one specific CPU.

The last two commands of Listing 3 show when bytecode is written to disk. Running the script directly compiles it in memory and writes nothing: the folder still holds the four files. Importing it as a module writes the bytecode to `__pycache__/period.cpython-313.pyc`, a cache that saves the compilation at the next import [9]. The file name carries the interpreter version, since the bytecode of 3.13 is not that of 3.12.

> The `.pyc` file is a cache, not the executable. It is written only when a module is imported. A script run as `python file.py` is compiled to bytecode in memory, and nothing appears on disk.

```
$ cat period.py
def period(freq):
    return 1 / freq


print(period(0.128))
$ python3.13 -m dis period.py
  ...
  5           LOAD_NAME                1 (print)
              PUSH_NULL
              LOAD_NAME                0 (period)
              PUSH_NULL
              LOAD_CONST               1 (0.128)
              CALL                     1
              CALL                     1
              POP_TOP
              RETURN_CONST             2 (None)

Disassembly of <code object period at 0x104f16250, file "period.py", line 1>:
  ...
  2           LOAD_CONST               1 (1)
              LOAD_FAST                0 (freq)
              BINARY_OP               11 (/)
              RETURN_VALUE
$ python3.13 period.py && ls -A
7.8125
C9.csv  period.py  predator_prey_cycles.ipynb  requirements.txt
$ python3.13 -c "import period" && ls __pycache__
7.8125
period.cpython-313.pyc
```

**Listing 3:** The bytecode of `period.py` under Python 3.13.11, abridged: the instructions that define the function at line 1 are left out. Running the script writes no cache. Importing it writes `__pycache__/period.cpython-313.pyc`.

### 1.4 Interpreters on disk and Python versions

**Definition 1.2:** The python command and PATH

An interpreter is a file on disk, and several can coexist on one machine. The **PATH** is the list of directories that the shell searches, in order, for the executable named on a command line. The command `python` runs the first executable of that name found along `PATH`, and `sys.executable` names the interpreter that is running [10]. An `import` searches the script's own folder, then the standard library and the `site-packages` of the running interpreter [11]. A package is therefore installed for one interpreter: the one whose `pip` installed it.

Listing 4 shows the interpreters of the lecture laptop under the four system directories of `PATH`. No file named `python` exists there. `python3` is the operating system's Python 3.9.6 in `/usr/bin`, and `python3.13` is Homebrew's 3.13.11, a link under `/opt/homebrew/bin` to the file named by `sys.executable`. The python.org installer puts its interpreter in `/usr/local/bin` instead. The names, paths and versions differ from machine to machine.

```
$ which -a python python3 python3.13
python not found
/usr/bin/python3
/opt/homebrew/bin/python3.13
$ python3 --version
Python 3.9.6
$ python3.13 --version
Python 3.13.11
$ python3.13 -c "import sys; print(sys.executable)"
/opt/homebrew/opt/python@3.13/bin/python3.13
```

**Listing 4:** The interpreters of the lecture laptop. `which -a` prints every match along `PATH`, and the reported version shows which one `python3` is.

The last sentence of Definition 1.2 is the cause of the most common error of the first weeks. Listing 5 shows two `pip` commands on the same laptop. The command `pip3` found on `PATH` belongs to the 3.9 interpreter, so a package installed with it is invisible to `python3.13`. The form `python3.13 -m pip` runs the `pip` module of the named interpreter and cannot install for another one.

> `pip` and `python` on one command line may belong to different interpreters. The form `python -m pip install` installs for the `python` that runs it. After a `ModuleNotFoundError` that follows an installation, `which python` and `python -m pip --version` name the two interpreters involved.

```
$ pip3 --version
pip 21.2.4 from /Library/Developer/CommandLineTools/Library/Frameworks/Python3.
    framework/Versions/3.9/lib/python3.9/site-packages/pip (python 3.9)
$ python3.13 -m pip --version
pip 25.3 from /opt/homebrew/lib/python3.13/site-packages/pip (python 3.13)
$ python3.13 -c "import numpy"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
    import numpy
ModuleNotFoundError: No module named 'numpy'
```

**Listing 5:** Two `pip` commands, two interpreters. `pip3` on `PATH` is the 3.9 interpreter's, and `python3.13 -m pip` is the 3.13 interpreter's. Neither has NumPy installed.

A minor version of Python appears every October and is supported for five years: about two years of bug-fix releases, then security fixes only [12]. Table 1 gives the status of the five supported versions in October 2026. Versions coexist on one machine as separate programs, `python3.9` and `python3.13` on the lecture laptop, and the venv of Section 2.1 fixes the version for a project. The course uses Python 3.13, because the scientific libraries support it fully. On Windows the course assumes WSL with Ubuntu [13]: every command of this chapter runs in the Ubuntu shell and applies unchanged.

> The newest release is not automatically the right choice. A compiled package such as NumPy needs a wheel built for the exact interpreter version, and in the months after a release some packages have none yet, so a version is chosen for the packages that must run on it. In October 2026 the packages used in this course, NumPy, SciPy, Matplotlib, pandas, JAX, Numba and scikit-learn, publish wheels for 3.13, 3.14 and 3.15 alike. The course uses 3.13 so that the lecture laptop, the notebooks and the students' machines run the same interpreter.

**Table 1:** The supported Python versions in October 2026 [12]. Each version is released in October and supported for five years.

| version | released | status in October 2026 | end of support |
|---|---|---|---|
| 3.11 | October 2022 | security fixes | October 2027 |
| 3.12 | October 2023 | security fixes | October 2028 |
| 3.13 | October 2024 | security fixes | October 2029 |
| 3.14 | October 2025 | bug fixes | October 2030 |
| 3.15 | October 2026 | bug fixes | October 2031 |

# 2 Virtual environments and packages

## 2.1 Virtual environments

**Definition 2.1:** Virtual environment

A **virtual environment**, or venv, is a directory created by `python -m venv` that gives one project its own `site-packages` [14]. It holds a file `pyvenv.cfg`, whose `home` key records the directory of the **base interpreter**, a `bin/python` that is a link to that interpreter, a `bin/pip` that installs into the venv, a script `bin/activate`, and an empty `lib/python3.X/site-packages`. The interpreter and the standard library are those of the base. The packages are the venv's own.

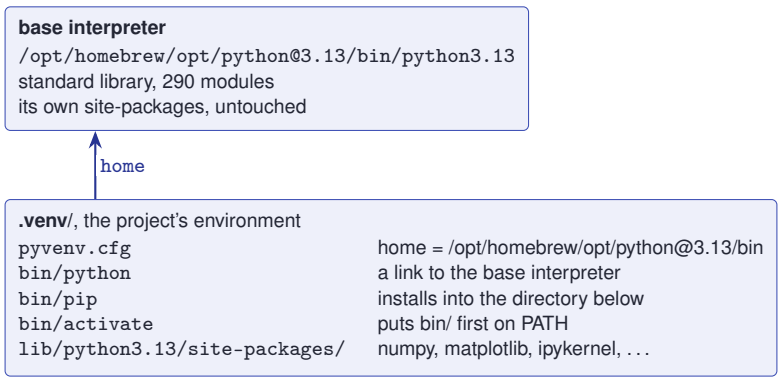



**Figure 2:** A virtual environment and its base interpreter. The venv owns a `site-packages` directory and links to the interpreter. It contains no copy of Python.

Listing 6 creates the environment of the project from the 3.13 interpreter and shows what

the directory holds. The key `home` of `pyvenv.cfg` and the link `bin/python3.13` both point at Homebrew's interpreter (Figure 2). The directory takes 12 MB before any package is installed, almost all of it the copy of `pip`.

> A venv is not a copy of the interpreter. Its `bin/python` is a link to the base interpreter named in `pyvenv.cfg`, so deleting or upgrading the base Python breaks every venv built from it. What the venv owns is its `site-packages` directory.

```
$ cd ~/attractor
$ python3.13 -m venv .venv
$ ls .venv
bin  include  lib  pyvenv.cfg
$ cat .venv/pyvenv.cfg
home = /opt/homebrew/opt/python@3.13/bin
include-system-site-packages = false
version = 3.13.11
executable = /opt/homebrew/Cellar/python@3.13/3.13.11/Frameworks/Python.framework/
    Versions/3.13/bin/python3.13
command = /opt/homebrew/opt/python@3.13/bin/python3.13 -m venv ~/attractor/.venv
$ ls -l .venv/bin/python*
lrwxr-xr-x  1 boris  wheel  10 20 Oct 23:09 .venv/bin/python -> python3.13
lrwxr-xr-x  1 boris  wheel  10 20 Oct 23:09 .venv/bin/python3 -> python3.13
lrwxr-xr-x  1 boris  wheel  44 20 Oct 23:09 .venv/bin/python3.13 -> /opt/homebrew/
    opt/python@3.13/bin/python3.13
$ du -sh .venv
 12M    .venv
```

**Listing 6:** A venv created from Python 3.13.11. `pyvenv.cfg` names the base interpreter, and the files under `bin/` named `python` are links to it.

A venv is built for one place. The `command` line of `pyvenv.cfg`, the first line of `bin/pip` and the `bin/activate` script all hold the absolute path of the directory, so a venv that is moved, copied to another machine or committed to a repository stops working (the first exercise on page 17 shows how). It is treated as disposable: a venv is deleted and rebuilt from `requirements.txt` whenever it is in doubt.

## 2.2 Activation, pip and requirements

The script `bin/activate` changes two things in the shell that sources it. It puts `.venv/bin` first on `PATH`, so that `python` and `pip` now mean the venv's copies, and it sets the variable `VIRTUAL_ENV`. The command `deactivate` undoes both. Listing 7 shows the `PATH` before and after, the interpreter that `python` now names, and the test that a program can apply to itself: inside a venv, `sys.prefix` is the venv and `sys.base_prefix` the base installation, so the two differ. Activation is a convenience for an interactive shell. A script or a scheduler runs the venv's interpreter directly, `.venv/bin/python script.py`, with the same effect.

`pip` installs packages from the Python Package Index, PyPI, into the `site-packages` of the interpreter that runs it [15, 16]. The file `requirements.txt` lists the packages a project needs, one per line, with a version constraint. Listing 8 installs the three packages of the project and the 35 packages they depend on, 207 MB in all, and checks the versions obtained. The next chapters on environments turn the three constraints into a lock file that fixes all 38 versions.

conda [17] is the main alternative to this pair of tools. It is both an environment manager and a package manager: a conda environment contains its own interpreter, and conda packages, from channels such as conda-forge, may contain non-Python software such as compilers, HDF5

```
$ echo $PATH
/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin
$ source .venv/bin/activate
(.venv) $ echo $PATH
~/attractor/.venv/bin:/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin
(.venv) $ echo $VIRTUAL_ENV
~/attractor/.venv
(.venv) $ which python
~/attractor/.venv/bin/python
(.venv) $ python --version
Python 3.13.11
(.venv) $ python -c "import sys; print(sys.prefix); print(sys.base_prefix)"
~/attractor/.venv
/opt/homebrew/opt/python@3.13/Frameworks/Python.framework/Versions/3.13
(.venv) $ python -c "import sys; print(sys.prefix != sys.base_prefix)"
True
```

**Listing 7:** Activation prepends `.venv/bin` to `PATH`. The venv's `python` is then the first found, and it reports a prefix different from its base.

```
(.venv) $ cat requirements.txt
numpy>=1.26
matplotlib>=3.8
ipykernel>=6.29
(.venv) $ python -m pip install -r requirements.txt
...
Successfully installed appnope-1.0.0 asttokens-3.0.2 comm-0.2.3 contourpy-1.4.0 ...
(.venv) $ python -c "import numpy, matplotlib, ipykernel; print(numpy.__version__,
    matplotlib.__version__, ipykernel.__version__)"
2.5.4 3.11.2 7.4.0
(.venv) $ du -sh .venv
207M    .venv
(.venv) $ deactivate
$ which python
python not found
```

**Listing 8:** Installing the project's requirements into the activated venv. The line `Successfully installed` names all 38 packages with their versions and is cut here after the first four.

or CUDA libraries. `pip install` inside a conda environment works, but conda does not record those packages, so a later `conda install` may overwrite them. The course uses `venv` and `pip` throughout.

> conda is a package and environment manager, not a library. It plays the roles of `pip` and `venv` together and also installs the interpreter itself. A library is code that a program imports, such as NumPy.

## 3 Notebooks and editors

## 3.1 Notebooks and kernels

**Definition 3.1:** Notebook, kernel and kernelspec

A **notebook** is a JSON file, extension `.ipynb`, holding cells and their outputs. A **front-end** such as JupyterLab or VS Code displays and saves it. A **kernel** is a separate Python process that executes the cells, with the packages of the environment whose `python` started it [18]. A **kernelspec** is a file `kernel.json` that records the path of that `python` and a display name, in a directory where every front-end looks (Figure 3).

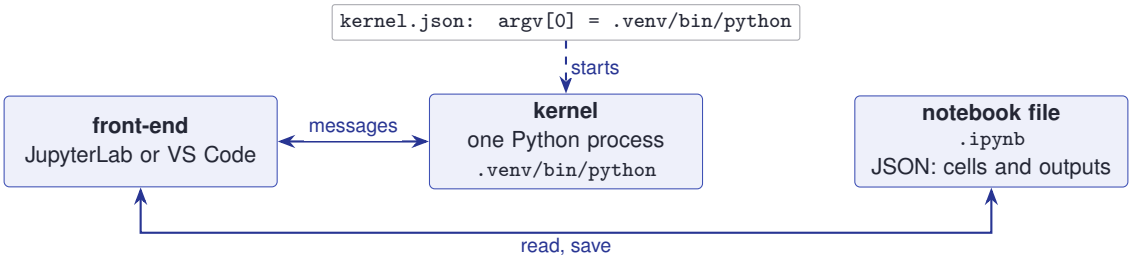



**Figure 3:** A front-end, a kernel and the notebook file. The front-end exchanges messages with the kernel and reads and saves the file. The kernelspec tells the front-end which `python` to start.

Cells run in the order in which they are executed, and the state of the kernel persists until it restarts. A notebook is reproducible when it runs from top to bottom from a fresh kernel, the command Restart Kernel and Run All Cells, which is how every notebook of the course is checked.

The package `ipykernel`, installed in the venv by `requirements.txt`, provides the kernel and writes the kernelspec (Listing 9). The option `--name` is the identifier and the default menu label, and a kernel of the same name is overwritten without a question. The option `--display-name "..."` sets another label. The option `--user` writes under the home directory (Table 2). The path in `kernel.json` is absolute, so a deleted venv breaks the kernel. A venv rebuilt at the same path with `ipykernel` installed serves the kernel again without a new registration, and `jupyter kernelspec uninstall attractor` removes a registration.

Activation applies to a shell, not to a kernel. A notebook runs on the kernel selected in the front-end from the list of registered kernelspecs. The venv's interpreter appears in that list once it is registered with `python -m ipykernel install --user --name attractor`, and activating the venv in a terminal changes nothing for the notebook.

**Table 2:** Where kernelspecs are written. The user directory is printed by `jupyter --data-dir`, followed by `kernels`.

| kernelspec | directory | written by |
|---|---|---|
| user kernels, macOS | `~/Library/Jupyter/kernels` | `ipykernel install --user` |
| user kernels, Linux | `~/.local/share/jupyter/kernels` | `ipykernel install --user` |
| the venv's own `python3` | `.venv/share/jupyter/kernels/python3` | `pip install ipykernel` |

## 3.2 JupyterLab and VS Code

JupyterLab [19] is a browser front-end. It is installed once, in a venv of its own, and reaches every project through the registered kernels, so no project needs it in its `requirements.txt`. Listing 10 creates that venv and starts JupyterLab on the notebook from the project folder. In the browser, the menu Kernel, then Change Kernel, offers `attractor`, and Restart Kernel and Run All Cells runs the notebook. Listing 11 is what the five code cells print. The first cell names

```
(.venv) $ python -m ipykernel install --user --name attractor
Installed kernelspec attractor in ~/Library/Jupyter/kernels/attractor
(.venv) $ head -4 ~/Library/Jupyter/kernels/attractor/kernel.json
{
 "argv": [
  "~/attractor/.venv/bin/python",
  "-Xfrozen_modules=off",
(.venv) $ jupyter kernelspec list
Available kernels:
  python3      ~/attractor/.venv/share/jupyter/kernels/python3
  attractor    ~/Library/Jupyter/kernels/attractor
(.venv) $ jupyter --data-dir
~/Library/Jupyter
```

**Listing 9:** Registering the venv as a kernel and reading the result. `kernel.json` holds the absolute path of the venv's `python`, and `jupyter kernelspec list` shows every registered kernel with its directory.

the venv's interpreter and confirms that it is a venv. The second loads 141 samples spanning 140 days. The third and fourth draw Figure 1. The last finds a dominant period of 7.8 days, close to the 8-day cycle of the nutrient supply, and a lag of about 2 days of the predator behind the prey, a quarter of the period.

```
$ python3.13 -m venv ~/venvs/jupyter
$ ~/venvs/jupyter/bin/python -m pip install jupyterlab
$ ~/venvs/jupyter/bin/jupyter lab --version
4.6.4
$ cd ~/attractor
$ ~/venvs/jupyter/bin/jupyter lab predator_prey_cycles.ipynb
```

**Listing 10:** JupyterLab in a venv of its own, created once, and started on the notebook from the project folder. The version is that of the lecture laptop.

```
~/attractor/.venv/bin/python
inside a venv: True
141 samples over 140 days
dominant period: 7.8 days (nutrient pulses every 4 days)
rotifers lag algae by about 2 days (a quarter period: 2.0 days)
```

**Listing 11:** The printed output of the five code cells of `predator_prey_cycles.ipynb` in the `attractor` kernel. The third and fourth cells draw Figure 1 and print nothing.

VS Code is an editor whose Python and notebook support comes from two extensions, Python (`ms-python.python`) and Jupyter (`ms-toolsai.jupyter`) [20, 21]. Every command is reached by name from the Command Palette, `Cmd+Shift+P` on macOS, `Ctrl+Shift+P` or `F1` elsewhere. Two choices of interpreter are made separately. `Python: Select Interpreter` sets the interpreter for scripts and the integrated terminal. Select Kernel, in an open notebook, offers three sources: Jupyter Kernels, the registered kernelspecs, Python Environments, the venvs that VS Code detects, `.venv` included, and Existing Jupyter Server. Only `ipykernel` is needed in the environment for either route. The integrated terminal runs the user's shell, a login shell on macOS and an interactive shell reading `~/.bashrc` on Linux. Environments are created with `python -m venv`, as in Listing 6, and not with the tool that VS Code may offer, so that the venv is the same whatever editor is used. The sequence is File, Open Folder, `attractor`, open the notebook, Select Kernel, Python Environments, `.venv`, then Run All. The output is that of

Listing 11.

# 4 Remote machines

## 4.1 Connecting to a remote machine over SSH

**Definition 4.1:** SSH connection and host key

**SSH** connects a client program on the laptop to a server on a remote machine through an encrypted channel, in which a terminal session, a file copy or a port forward can run. The server proves its identity with a **host key**, whose **fingerprint** the client shows at the first connection and stores in `~/.ssh/known_hosts`. The user proves theirs with a password or a private key, and some sites add a second factor, a one-time code. The file `~/.ssh/config` gives a host a short name and fixes the options used for it [22].

At the first connection the client does not know the server and asks (Listing 12). The fingerprint identifies the server, and it must equal the value that the site publishes, for CSD3 on its host keys page [23]. The answer is `yes`, typed in full, and the key is then stored and never asked again. A later warning that the host key has changed means either that the site replaced its key, which it announces, or that the connection does not reach the expected server.

```
$ ssh <crsid>@login-icelake.hpc.cam.ac.uk
The authenticity of host 'login-icelake.hpc.cam.ac.uk' can't be established.
ED25519 key fingerprint is SHA256:nFVSXK+VRGCaUupQEdhXzO6kp01m2fzzmbgPr0sc2so.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

**Listing 12:** The prompt of the first connection to a server. The fingerprint shown is the one published for every `login*.hpc.cam.ac.uk` node of CSD3 in October 2026.

Listing 13 is the configuration block of the course. `Host csd3` names the block, and the name typed after `ssh` is matched against it, so `ssh csd3` replaces the full command of Listing 12, with its user name and host name. The three `Control` lines share one connection: with `ControlMaster auto`, the first `ssh` to the host opens a master connection and every later `ssh`, `scp` or editor connection to the same host travels inside it, so the password and the one-time code are asked once. `ControlPersist 900` keeps the master open for 900 s after the last session closes, and a new login is needed only after that. The command `ssh -fN csd3` opens the shared connection in the background without a terminal.

```
Host csd3
    HostName login-icelake.hpc.cam.ac.uk
    User <crsid>
    ControlMaster auto
    ControlPath ~/.ssh/control-%C
    ControlPersist 900
```

**Listing 13:** The block of `~/.ssh/config` for the remote machine. The options apply to every command that names the host `csd3`.

Listing 14 logs in, creates the project folder on the remote machine and copies the project files to it with `scp`, which takes the same `host:path` form as `ssh`. The password and the code are asked at the first command only. The form `ssh csd3 ls attractor` runs one command on the remote machine and returns. Later chapters replace the copy by `git clone`.

```
$ ssh csd3
Password:
Verification code:
[<crsid>@login-q-2 ~]$ hostname
login-q-2
[<crsid>@login-q-2 ~]$ mkdir attractor; exit
$ scp requirements.txt C9.csv predator_prey_cycles.ipynb csd3:attractor/
$ ssh csd3
[<crsid>@login-q-2 ~]$ ls ~/attractor
C9.csv  predator_prey_cycles.ipynb  requirements.txt
```

**Listing 14:** Login with password and one-time code, a folder created on the remote machine, and the project files copied into it through the shared connection. CSD3, October 2026.

### 4.2 Python environments on a remote machine

**Definition 4.2:** Environment module

On a shared Linux machine, compilers, libraries and interpreters are offered as **environment modules** [24]. `module avail python` lists the Python modules, `module load <name>` puts one on the `PATH` of the current shell, and `module list` and `module purge` show and unload what is loaded. A module changes the shell that loaded it and nothing else.

The venv is rebuilt on the remote machine, from the same `requirements.txt`, with the interpreter of the module (Listing 15). It is never copied from the laptop: its paths are absolute, and the compiled packages it holds were built for the laptop's processor and operating system. The module is loaded in the shell that needs it, not in `~/.bashrc`, so that every shell and every kernel starts from a known state. The kernel is registered with the same command as on the laptop, and the kernelspec lands in the Linux directory of Table 2. The module of the course, `python/3.11.0-icl`, is Python 3.11.0 built for the Ice Lake login nodes, checked in October 2026. Module names change as the software is rebuilt, and `module avail python` is the reference.

```
[<crsid>@login-q-2 ~]$ module avail python
[<crsid>@login-q-2 ~]$ module load python/3.11.0-icl
[<crsid>@login-q-2 ~]$ python --version
Python 3.11.0
[<crsid>@login-q-2 ~]$ python -m venv ~/venvs/attractor
[<crsid>@login-q-2 ~]$ source ~/venvs/attractor/bin/activate
(attractor) $ python -m pip install -r ~/attractor/requirements.txt
(attractor) $ python -m ipykernel install --user --name attractor
Installed kernelspec attractor in /home/<crsid>/.local/share/jupyter/kernels/
    attractor
```

**Listing 15:** The same `requirements.txt`, a different interpreter and machine: a venv built on the CSD3 login node from the Python module, and its kernel registered. CSD3, October 2026.

### 4.3 VS Code Remote-SSH

The extension Remote - SSH (`ms-vscode-remote.remote-ssh`) [25] splits VS Code in two (Figure 4). The window stays on the laptop. A VS Code Server, installed in `~/.vscode-server` on the remote machine, opens the files, runs the terminals and starts the kernels there. The Python and Jupyter extensions are installed a second time, on the remote side. The host is the alias of

Listing 13, so the shared connection is reused.

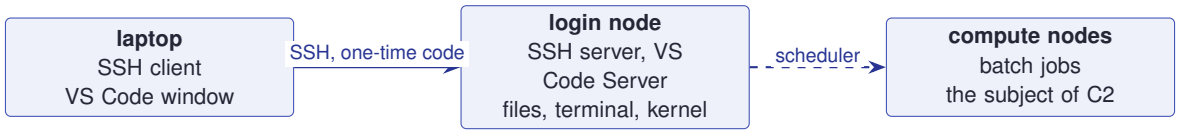



**Figure 4:** Working on a remote machine. The laptop holds the SSH client and the VS Code window. The login node runs the server side of both, and the compute nodes are reached from it by a scheduler.

The sequence of the demonstration is the following.

1. In the Extensions view, install Remote - SSH. In the settings, tick `Remote.SSH: Show Login Terminal`, so that the password and code prompts appear in a terminal.
2. Command Palette, `Remote-SSH: Connect to Host...`, `csd3`. Enter the password and the code in the terminal that appears.
3. The bottom-left corner shows `SSH: csd3`. Open Folder `/home/<crsid>/attractor`, the folder of Listing 14, and install Python and Jupyter "in SSH: csd3".
4. Open `predator_prey_cycles.ipynb`, Select Kernel, Jupyter Kernels, `attractor`, or Python Environments, `~/venvs/attractor/bin/python`, then Run All.
5. The first cell prints `/home/<crsid>/venvs/attractor/bin/python`, and the figure is identical to the laptop's.

A finished session is closed with File, Close Remote Connection. Server processes that linger on the login node are stopped with `Remote-SSH: Kill VS Code Server on Host`. When Remote-SSH is unavailable, a JupyterLab server started on the remote machine can be reached through an SSH port forward.

VS Code opens two SSH connections per window. Through a load-balanced address they can reach two different login nodes, and each asks for a code. The remedies, in order, are the setting `remote.SSH.showLoginTerminal` set to `true`, the `ControlMaster` options of Listing 13, and the setting `"remote.SSH.useLocalServer": false`.

> A host name that balances over several machines is several servers. Two connections opened one after the other can land on different nodes, and a shared connection, `ControlMaster auto`, is what pins every later connection to the node of the first.

## Troubleshooting

Eight symptoms point back to the concepts of this chapter.

| symptom | cause | remedy |
|---|---|---|
| `ModuleNotFoundError` right after a successful `pip install`. | `pip` and `python` belong to different interpreters (Definition 1.2, Listing 5). | Activate the venv and use `python -m pip install`. `which python` names the interpreter. |
| `python: command not found` on macOS outside a venv. | The system provides `python3` and `python3.X` only (Listing 4). | Activate the venv, or type `python3.13`. |
| `bad interpreter` from `.venv/bin/pip`, or `which python` finds nothing after activation. | The venv was moved or renamed and holds absolute paths (Section 2.1). | Delete the venv and rebuild it from `requirements.txt`. |
| The kernel `attractor` is not in the list. | The kernelspec was not written (Definition 3.1). | Run `python -m ipykernel install --user --name attractor` with the venv's `python`. |
| The kernel is listed but fails to start. | `kernel.json` points to a deleted or recreated venv (Listing 9). | Register the kernel again, or remove it with `jupyter kernelspec uninstall`. |
| VS Code offers no kernel for `.venv`. | `ipykernel` is missing from the venv, or the Python and Jupyter extensions are missing, on the remote side for Remote-SSH (Section 3.2). | Install `ipykernel` in the venv, install the extensions, then `Developer: Reload Window`. |
| The first `ssh` shows a fingerprint and asks for `yes`. | The client does not know the server yet (Definition 4.1). | Compare the fingerprint with the published one, then answer `yes`. |
| Remote-SSH asks for the one-time code again and again. | Two connections reach different nodes behind the balanced address (Section 4.3). | Show the login terminal, add the `ControlMaster` options, set `useLocalServer` to false. |

## Key points

- `python` is the first executable of that name along `PATH`, and a package belongs to the interpreter whose `pip` installed it (Definition 1.2).
- A venv is a directory with a link to its base interpreter and a private `site-packages` (Definition 2.1). Activation only edits `PATH` (Listing 7). A venv is disposable and is rebuilt from `requirements.txt`, never moved or copied.
- A notebook runs in a kernel, a separate process named by a kernelspec that records an absolute `python` path (Definition 3.1). `ipykernel` registers a venv, and a notebook is checked by a run from top to bottom on a fresh kernel.
- SSH identifies the server by its host key fingerprint and the user by a password or key (Definition 4.1). A `Host` block with `ControlMaster` shares one authenticated connection among terminals, `scp` and the editor (Listing 13).
- On a remote machine the interpreter comes from a module (Definition 4.2), and the venv and its kernel are rebuilt there from the same `requirements.txt` (Listing 15).

## Course environment

This section is specific to the course of Michaelmas 2026 and will not be found in a general text.

### The course website

All material lives at `https://teaching.borisbolliet.io/c1-2026/` [2], one page per lecture with the summary, the contents, the slides and the notes. Slides and notes appear before each lecture, recordings and transcripts after it. The course assistant is a chat that answers from the course material, with its sources listed under each answer. Its Self-test tab offers multiple-choice questions per released lecture, with the answer and an explanation. The login is the CRSid and the course password given in the lectures. The site also offers a calendar subscription and the page of the example classes, where the exercises of the course are set.

### Coding assistants

Any coding assistant may be used for the example classes and the coursework, as in research. The popular choices in 2026 are Claude Code, Codex, Cursor and Grok Bot [26]. Every submitted line, and every command run to produce it, must be explainable by the student. No command is pasted unless its effect can be predicted, and the live demonstrations are typed by hand.

### Python and tools

The course uses Python 3.13, one venv per project created with `venv` and filled with `pip`, JupyterLab or VS Code as the front-end, and CPython throughout. On Windows the course assumes WSL with Ubuntu [13].

### CSD3

CSD3 stands for Cambridge Service for Data Driven Discovery. It is the University of Cambridge's high-performance computing cluster, operated by Research Computing Services and documented at `https://docs.hpc.cam.ac.uk/` [27]. It runs Rocky Linux 8 with `bash`, and the username is the CRSid. An account requires the HPC Account Application form [28] before any login attempt, and the route for this MPhil is announced by the course organisers. A CRSid alone gives no access.

### Login nodes and storage

The login nodes `login-q-1` to `login-q-4` serve for editing, small tests and environment set-up. They are shared, and a watchdog stops long or large processes. Batch jobs on the compute nodes are the subject of the C2 module. The address `login-icelake.hpc.cam.ac.uk` balances over the four Ice Lake login nodes, and `login-cpu.hpc.cam.ac.uk` over all login nodes, including the Cascade Lake nodes `login-p-1` to `login-p-4` [29]. Code lives in `/home/<crsid>`, 50 GB with snapshots in `.snapshot`. Data live in `/rds/user/<crsid>/hpc-work`, linked as `~/rds`, 1 TB without backup, and `quota` shows the usage [30].

### Multi-factor authentication

Every login asks for the password, or an SSH key, and a 6-digit time-based one-time code from an authenticator application [31]. The code is set up once, before the first login, in a terminal on `multi.hpc.cam.ac.uk`, which prints a QR code for Google Authenticator or Microsoft Authenticator, account type "Other" (Listing 16). The entry appears as "CSD3: SSH Login", and the emergency codes shown afterwards are kept. Three failed attempts in a

row, on `multi.hpc.cam.ac.uk` or on a login node, block the client address for 20 minutes, so retries stop after the second failure. A code that is always rejected may come from the Login-Web entry, "CSD3: <crsid>", instead of "CSD3: SSH Login". The Login-Web portal, https://login-web.hpc.cam.ac.uk, with its own code entry, is the alternative to Remote-SSH on CSD3, and the Jupyter page of the CSD3 documentation [32] describes the port-forwarded server.

```
$ export LANG=en_GB.UTF-8            # macOS: avoids a garbled QR code
$ ssh <crsid>@multi.hpc.cam.ac.uk    # University password, scan QR code
$ ssh <crsid>@login-icelake.hpc.cam.ac.uk
Password:
Verification code:                   # 6 digits from "CSD3: SSH Login"
[<crsid>@login-q-2 ~]$ hostname
login-q-2
```

**Listing 16:** One-time set-up of the second factor on CSD3 and the first login. CSD3, October 2026.

## Exercises

Solutions are given from page 18.

### A moved virtual environment

The venv of Listing 6 is renamed with `mv .venv .venv-moved`. For each of the following commands, predict whether it works and what it prints, then explain the outcome from the contents of the venv: (a) `.venv-moved/bin/python -c "import sys; print(sys.prefix)"`, (b) `.venv-moved/bin/pip --version`, (c) `source .venv-moved/bin/activate` followed by `which python`. (d) What is the right way to move a project's environment to another folder or another machine?

### The interpreter behind a command

On the lecture laptop, with the `PATH` of Listing 4, a student runs `pip3 install numpy` and then `python3.13 -c "import numpy"`, which fails with `ModuleNotFoundError`. (a) Which interpreter received NumPy, and why? (b) Which two commands show the mismatch? (c) Why does the venv of Section 2.1 remove the problem whatever the `PATH`?

### A kernel whose environment has gone

After the registration of Listing 9, the venv is deleted with `rm -rf .venv` and rebuilt with `python3.13 -m venv .venv` and `pip install -r requirements.txt`. (a) Does `jupyter kernelspec list` still show `attractor`? (b) Does the kernel start in JupyterLab before the rebuild, and after it? (c) Which file decides, and which command repairs the registration?

## Solutions to the exercises

### A moved virtual environment

(a) It works and prints the new location. `bin/python` is a link to the base interpreter, an absolute path that did not change, and the interpreter finds `pyvenv.cfg` next to itself, so `sys.prefix` is `.venv-moved`. (b) It fails with `bad interpreter`. The first line of `bin/pip` names the old `.venv/bin/python3.13`, which no longer exists. (c) Activation succeeds, but `which python` finds nothing: the script exports the old path of the venv in `VIRTUAL_ENV` and puts the old, absent `bin` directory first on `PATH`. (d) The venv is not moved. The project files and `requirements.txt` are moved or copied, and a new venv is created there with `python -m venv` and `python -m pip install -r requirements.txt`, as in Listing 15 for the remote machine.

**Answer check**

```
$ mv .venv .venv-moved
$ .venv-moved/bin/python -c "import sys; print(sys.prefix)"
~/attractor/.venv-moved
$ .venv-moved/bin/pip --version
zsh:1: .venv-moved/bin/pip: bad interpreter: ~/attractor/.venv/bin/python: no
    such file or directory
$ head -1 .venv-moved/bin/pip
#!~/attractor/.venv/bin/python3.13
$ source .venv-moved/bin/activate
(.venv) $ echo $VIRTUAL_ENV
~/attractor/.venv
(.venv) $ which python
python not found
```

### The interpreter behind a command

(a) The 3.9 interpreter in `/usr/bin`. The only `pip3` on `PATH` is `/usr/bin/pip3`, which belongs to it, and `pip` installs into the `site-packages` of the interpreter that runs it (Definition 1.2). (b) `pip3 --version` and `python3.13 -m pip --version`, which name their interpreters in parentheses, as in Listing 5. (c) Inside the activated venv, `python` and `pip` are both `.venv/bin` entries that resolve to the same interpreter, and `python -m pip` installs into the venv's `site-packages` even when nothing is activated.

**Answer check**

```
$ which pip3
/usr/bin/pip3
$ pip3 --version
pip 21.2.4 from /Library/Developer/CommandLineTools/Library/Frameworks/Python3.
    framework/Versions/3.9/lib/python3.9/site-packages/pip (python 3.9)
$ python3.13 -m pip --version
pip 25.3 from /opt/homebrew/lib/python3.13/site-packages/pip (python 3.13)
(.venv) $ python -m pip --version
pip 25.3 from ~/attractor/.venv/lib/python3.13/site-packages/pip (python 3.13)
```

### A kernel whose environment has gone

(a) Yes. The kernelspec lives under `~/Library/Jupyter/kernels`, outside the venv, and deleting the venv does not touch it. The list changes only for the venv's own `python3` entry, which

disappears with the venv. (b) Before the rebuild the start fails, because `kernel.json` names `.venv/bin/python`, which no longer exists. After the rebuild it starts again, because the new venv has the same path and `ipykernel` was reinstalled from `requirements.txt`. A venv rebuilt elsewhere, or without `ipykernel`, keeps failing. (c) `kernel.json` decides, through the first element of its `argv`. The registration is repaired by running `python -m ipykernel install --user --name attractor` again from the new venv, which overwrites the file.

**Answer check**

The start of the kernel is attempted here from the JupyterLab venv with the library that JupyterLab itself uses.

```
$ rm -rf .venv
$ ~/venvs/jupyter/bin/jupyter kernelspec list
Available kernels:
  python3      ~/venvs/jupyter/share/jupyter/kernels/python3
  attractor    ~/Library/Jupyter/kernels/attractor
$ ~/venvs/jupyter/bin/python -c "from jupyter_client import KernelManager;
    KernelManager(kernel_name='attractor').start_kernel()"
...
FileNotFoundError: [Errno 2] No such file or directory: '~/attractor/.venv/bin/
    python'
```

## References


[1] 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.

[2] Boris Bolliet. C1 Research Computing and Software Development, Michaelmas 2026: course website. https://teaching.borisbolliet.io/c1-2026/. accessed 20 October 2026.

[3] Chemostat. Wikipedia, https://en.wikipedia.org/wiki/Chemostat. accessed 20 October 2026.

[4] Python Software Foundation. CPython: the Python programming language. Source repository, https://github.com/python/cpython. accessed 20 October 2026.

[5] Python Software Foundation. Glossary: bytecode. Python 3.13 documentation, https://docs.python.org/3/glossary.html#term-bytecode. accessed 20 October 2026.

[6] Python Software Foundation. The Python standard library. Python 3.13 documentation, https://docs.python.org/3/library/index.html. accessed 20 October 2026.

[7] PyPy. Wikipedia, https://en.wikipedia.org/wiki/PyPy. accessed 20 October 2026.

[8] Python Software Foundation. `dis`: disassembler for Python bytecode. Python 3.13 documentation, https://docs.python.org/3/library/dis.html. accessed 20 October 2026.

[9] Python Software Foundation. Modules: "compiled" Python files. Python 3.13 tutorial, https://docs.python.org/3/tutorial/modules.html#compiled-python-files. accessed 20 October 2026.

[10] Python Software Foundation. Using the Python interpreter. Python 3.13 tutorial, https://docs.python.org/3/tutorial/interpreter.html. accessed 20 October 2026.

[11] 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 20 October 2026.

[12] Python Software Foundation. Status of Python versions. Python Developer's Guide, https://devguide.python.org/versions/. accessed 20 October 2026.

[13] Microsoft. How to install Linux on Windows with WSL. https://learn.microsoft.com/en-us/windows/wsl/install. accessed 20 October 2026.

[14] Python Software Foundation. `venv`: creation of virtual environments. Python 3.13 documentation, https://docs.python.org/3/library/venv.html. accessed 20 October 2026. The section "How venvs work" describes `pyvenv.cfg`.

[15] pip developers. pip user guide. https://pip.pypa.io/en/stable/user_guide/. accessed 20 October 2026.

[16] Python Packaging Authority. Install packages in a virtual environment using pip and venv. Python Packaging User Guide, https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/. accessed 20 October 2026.

[17] conda contributors. conda documentation. https://docs.conda.io/projects/conda/en/stable/. accessed 20 October 2026. Packages from https://conda-forge.org/.

[18] Jupyter Development Team. IPython kernel documentation. https://ipykernel.readthedocs.io/en/stable/. accessed 20 October 2026. Kernel registration: https://ipython.readthedocs.io/en/stable/install/kernel_install.html.

[19] Jupyter Development Team. JupyterLab documentation. https://jupyterlab.readthedocs.io/en/stable/. accessed 20 October 2026.

[20] Microsoft. Jupyter notebooks in VS Code. https://code.visualstudio.com/docs/datascience/jupyter-notebooks. accessed 20 October 2026.

[21] Microsoft. Python environments in VS Code. https://code.visualstudio.com/docs/python/environments. accessed 20 October 2026.

[22] OpenBSD. `ssh_config`: OpenSSH client configuration file. OpenBSD manual pages, https://man.openbsd.org/ssh_config. accessed 20 October 2026. `ControlMaster`, `ControlPath` and `ControlPersist`.

[23] University of Cambridge Research Computing Services. SSH host keys. CSD3 documentation, https://docs.hpc.cam.ac.uk/hpc/user-guide/hostkeys.html. accessed 20 October 2026.

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

[25] Microsoft. Remote development using SSH. https://code.visualstudio.com/docs/remote/ssh. accessed 20 October 2026.

[26] Coding assistants in use in 2026: Claude Code, Codex, Cursor and Grok Bot. https://claude.com/claude-code, https://openai.com/codex/, https://cursor.com/, https://x.ai/bot. accessed 20 October 2026.

[27] University of Cambridge Research Computing Services. CSD3 documentation. https://docs.hpc.cam.ac.uk/. accessed 20 October 2026.

[28] University of Cambridge Research Computing Services. HPC account application. https://www.hpc.cam.ac.uk/rcs-application. accessed 20 October 2026.

[29] University of Cambridge Research Computing Services. Connecting to CSD3. CSD3 documentation, https://docs.hpc.cam.ac.uk/hpc/user-guide/connecting.html. accessed 20 October 2026.

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

[31] University of Cambridge Research Computing Services. Multi-factor authentication. CSD3 documentation, https://docs.hpc.cam.ac.uk/hpc/user-guide/mfa.html. accessed 20 October 2026.

[32] University of Cambridge Research Computing Services. Jupyter on CSD3. CSD3 documentation, https://docs.hpc.cam.ac.uk/hpc/software-packages/jupyter.html. accessed 20 October 2026.

## Index

base interpreter, 6
built-ins, 3
bytecode, 3

CPython, 3

environment modules, 12

fingerprint, 11
front-end, 9

host key, 11

kernel, 9
kernelspec, 9

notebook, 9

PATH, 5
Python interpreter, 3

SSH, 11
standard library, 3

third-party packages, 3

virtual environment, 6