Lecture 1 — Course environment

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

Boris Bolliet

University of Cambridge

Outline

- Research computing in the age of AI coding assistants
- The course website and the running project
- The Python interpreter, Python versions and virtual environments
- Notebooks, kernels, JupyterLab and VS Code
- CSD3: SSH, multi-factor authentication, VS Code Remote-SSH
- Live demonstrations throughout

The researcher as reviewer

- Predicting the effect of a shell command.
- 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 a computation's cost beforehand.

Key point

Every submitted line must be explainable by its author, whatever tool wrote it.

Policy on assistants

- Any coding assistant may be used, as in research; popular choices: Claude Code, Codex, Cursor, Grok Bot.
- Every submitted line and every command run must be explainable.
- No command is pasted unless its effect can be predicted.
- The live demonstrations are typed by hand.

The course website

https://teaching.borisbolliet.io/c1-2026/

- One page per lecture: summary, contents, slides, notes.
- Slides and notes before each lecture; recordings and transcripts after.
- **Course assistant**: a chat answering from the course material, and a **self-test** of multiple-choice questions per lecture.
- Login with your CRSid and the course password.
- Calendar subscription and example classes page.

The running project: `attractor`

- One Python package, `attractor`, grown over the term, covering some basics of dynamical systems.
- Today's example notebook and dataset: a predator–prey time series, rotifers eating algae in a chemostat (Blasius *et al.*, *Nature* 2020; data on figshare).
- The notebook is only a vehicle for the tools of this lecture.

Goal

By the end of term we have built together a scientific Python package.

✎ The project folder

```
$ 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
$ ls
C9.csv  period.py  predator_prey_cycles.ipynb  requirements.txt
```

- The same four files are under *Demonstration files* on the lecture page, with a viewer and a Colab link for the notebook.
- `unzip -j` drops the folder inside the archive.

Inside the Python interpreter

- **CPython**, the reference interpreter, is a program written in C.
- It **compiles** source to bytecode (cached in `__pycache__/*.pyc` for imported modules), then **executes** it.
- Execution: a loop in C that carries out the bytecode instructions one by one.
- **Standard library**: the modules shipped with the interpreter (`os`, `json`, `math`, ...).
- `sys.stdlib_module_names` lists them; `python -m dis script.py` shows bytecode.
- PyPy is another implementation, with a just-in-time compiler; the course uses CPython.

Using the interpreter (tutorial)

✎ Bytecode

`period.py`: `def period(freq): return 1 / freq` and `print(period(0.128))`

```
$ python3.13 -m dis period.py          # abridged
  5           LOAD_NAME                1 (print)
              LOAD_NAME                0 (period)
              LOAD_CONST               1 (0.128)
              CALL                     1
Disassembly of <code object period ...>:
  2           LOAD_CONST               1 (1)
              LOAD_FAST                0 (freq)
              BINARY_OP               11 (/)
              RETURN_VALUE
$ python3.13 -c "import period" && ls __pycache__
7.8125
period.cpython-313.pyc
```

- Each line is one instruction carried out by the interpreter; the `.pyc` caches them. Example output from Python 3.13; names differ between versions.

The Python interpreter on disk

- The interpreter is a file on disk (`/usr/bin/python3`); several can coexist.
- `python` runs the first match found along `PATH`.
- `import` searches the script's folder, then that interpreter's standard library and `site-packages`.
- `sys.executable` names the running interpreter.
- Windows: the course assumes WSL (Ubuntu); every command then applies unchanged.

Key point

A package is installed for one interpreter: the one whose `pip` installed it.

Python versions

- A minor version each October, supported five years.
- Several versions coexist as separate interpreters, e.g. `python3.9` and `python3.13` on the lecture laptop.

Key point

The course uses Python 3.13 this year. Each project has a venv of its own, created from a chosen interpreter.

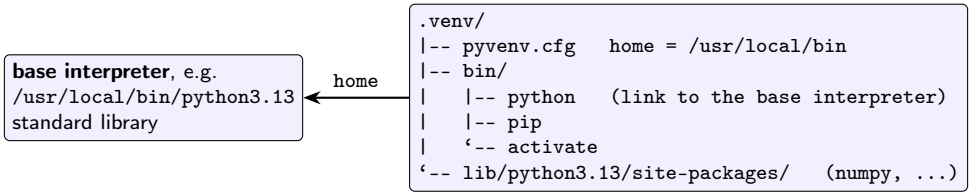
python.org downloads

✎ Interpreters on the laptop

```
$ which -a python3 python3.13
/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
$ python3.13 -c "import sys; print(len(sys.stdlib_module_names))"
290
```

- On the lecture laptop: the operating system's 3.9 and Homebrew's 3.13; `python3` is the first on `PATH`.
- Example output: on another machine the names, paths, versions and counts differ.

Virtual environments



- A **venv** is a directory: a base-interpreter link plus a private `site-packages`.
- Created by `python -m venv .venv`; disposable, never committed.
- Rebuilt anywhere from `requirements.txt`.

Key point

Each project has its own venv, created with `venv` and filled with `pip`.

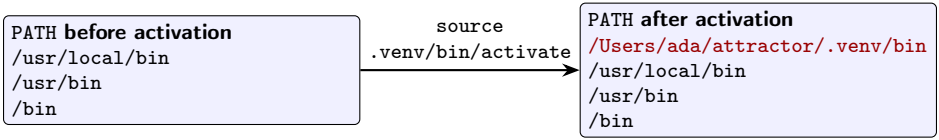
conda versus pip

- `pip` installs Python packages from PyPI into an existing interpreter; environments come from `venv`.
- `conda` creates environments and installs the interpreter itself and non-Python libraries (compilers, HDF5, CUDA) from channels such as conda-forge.
- `pip` inside a conda environment works; `conda` does not track what `pip` installed.

Key point

`venv` and `pip` throughout the course.

Activation and PATH



- `source .venv/bin/activate` only puts the venv's `bin` first in `PATH` (and sets `VIRTUAL_ENV`); `deactivate` removes it.
- `which python` names the winner.

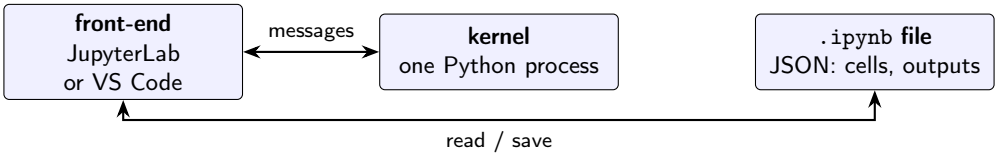
pip user guide

✎ The project venv

```
$ cd ~/attractor
$ python3.13 -m venv .venv
$ source .venv/bin/activate
(.venv) $ which python
/Users/ada/attractor/.venv/bin/python
(.venv) $ python --version
Python 3.13.11
(.venv) $ python -m pip install -r requirements.txt
(.venv) $ python -c "import sys; print(sys.prefix != sys.base_prefix)"
True
```

- `requirements.txt`: numpy, matplotlib, ipykernel.
- `deactivate` restores the previous `PATH`.
- Example output: the path shows the project folder of the machine in use.

Notebooks and kernels



- A notebook is a JSON file of cells; the **kernel** executes them.
- The kernel is one Python interpreter, hence one venv.
- Cells run in the order clicked; state persists until restart.

Key point

A notebook is reproducible when it runs top to bottom from a fresh kernel (Restart Kernel and Run All).

JupyterLab documentation | ipykernel

Kernel registration: `ipykernel`

- A **kernelspec** records a `python` path and a name.
- Inside the venv: `pip install ipykernel`.
- `python -m ipykernel install --user --name attractor` registers it.
- `jupyter kernelspec list` shows every kernel and its path.
- JupyterLab itself is installed once, in a venv of its own (`pip install jupyterlab`), not in each project.

Key point

JupyterLab lists a venv once its kernelspec exists; VS Code also detects `.venv` folders directly.

JupyterLab documentation

✎ The kernel and the notebook

```
(.venv) $ python -m ipykernel install --user --name attractor
Installed kernelspec attractor in /Users/ada/Library/Jupyter/kernels/attractor
(.venv) $ jupyter kernelspec list
$ python3.13 -m venv ~/venvs/jupyter                        # once
$ ~/venvs/jupyter/bin/python -m pip install jupyterlab    # once
$ ~/venvs/jupyter/bin/jupyter lab predator_prey_cycles.ipynb
```

- JupyterLab: Kernel → Change Kernel → `attractor`; Restart Kernel and Run All.
- The first cell prints `sys.executable`; the last cell, the dominant period and the predator's lag.
- VS Code: open the folder, Select Kernel → `.venv`, Run All.

VS Code

- A **front-end** like JupyterLab: it starts the interpreter and kernel processes that do the work.
- Two interpreter choices: one for scripts, one kernel for notebooks.
- Its integrated terminal runs the user's shell (a login shell on macOS).
- The Python and Jupyter extensions are required.

Key point

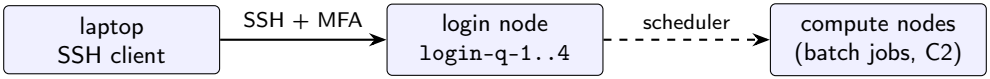
Selecting a kernel sets which interpreter runs the notebook.

CSD3 account application

- An HPC account is required before connecting to CSD3.
- Cambridge users must submit the HPC Account Application before attempting to log in.
- A CRSid alone does not provide CSD3 access.

CSD3 Quick Start

CSD3: Cambridge HPC service



- Rocky Linux 8, `bash`; the account name is the CRSid.
- **Login nodes**: editing, small tests, environments; a watchdog stops longer work.
- Code in `/home` (backed up); data in `~/rds` (1 TB, no backup).
- Software from **modules**: `module avail python`, `module load <name>`.

CSD3 documentation

SSH and multi-factor authentication

- **SSH**: client, server, encrypted channel.
- CSD3 documents `login-cpu.hpc.cam.ac.uk` for CPU access; `login.hpc.cam.ac.uk` is an alias. The course uses `login-icelake.hpc.cam.ac.uk` to load `python/3.11.0-icl`, the Python module for Ice Lake nodes.
- **MFA**: password plus a 6-digit TOTP code, set up once on `multi.hpc.cam.ac.uk`.
- A shared connection in `~/.ssh/config`: MFA is asked once, then reused until 900 s after the last session closes.

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

✎ Logging in and copying the project

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

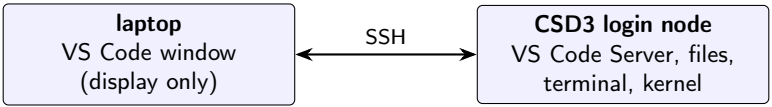
- The password and the TOTP code are asked once; `scp` reuses the connection.
- `ssh csd3 ls attractor` runs one command and returns.

✎ A venv on CSD3

```
[<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
```

- The same `requirements.txt`, a different interpreter and machine.
- `module avail python` lists the alternatives.

VS Code Remote-SSH



- Remote-SSH installs a server in `~/.vscode-server`; only the display stays local.
- Python and Jupyter extensions are installed again on the remote side.
- MFA prompts: notes, section 11.

Key point

A remote window edits, runs terminals and starts kernels on CSD3.

Jupyter on CSD3

✎ The notebook on CSD3 from VS Code

1. Command Palette → `Remote-SSH: Connect to Host...` → `csd3`
2. Open Folder `/home/<crsid>/attractor`; install the Python and Jupyter extensions "in SSH: csd3"
3. Open `predator_prey_cycles.ipynb`, Select Kernel → `attractor`, Run All
4. The first cell prints `/home/<crsid>/venvs/attractor/bin/python`; the figure is the laptop's

Summary

- Website: slides and notes before, recordings after.
- Interpreters are found through `PATH`; packages belong to one interpreter.
- A venv per project, disposable, rebuilt from `requirements.txt`.
- A notebook runs in a kernel; `ipykernel` registers a venv.
- CSD3 over SSH with MFA; Remote-SSH runs kernels there.