

Modules, Packages, Virtual Environments and Git

Organise code into modules and packages, install libraries with pip or uv inside virtual environments, pin dependencies, use the standard library, and version-control your projects with Git and GitHub.

● Python class 10

✦ Python 3.13 · pip 25 · uv · Git 2.5x

Intermediate

🕒 90 min

By the end of this lesson you can

- Import modules in the different ways and explain if **name** == "main"
- Create a package with **init.py** and a sensible project layout
- Create virtual environments and install and pin packages with pip or uv
- Use key standard-library modules (os, pathlib, datetime, random, math, statistics, collections)
- Use the everyday Git workflow: init, add, commit, branch, push, pull request

Modules

A **module** is any `.py` file. Importing it gives you its functions, classes and variables.

Python

```
import math                                # whole module, use with prefix
from statistics import mean, median        # specific names
import datetime as dt                      # alias
from collections import Counter as C
print(math.sqrt(144), math.pi, mean([3, 5, 10]), median([3, 5, 10]), dt.date(2025, 4,
1).strftime("%d %b %Y"), C("aab"))
print(math.ceil(4.2), math.floor(4.8), math.log10(1000), round(math.exp(1), 4),
math.comb(5, 2))
```

OUTPUT

```
12.0 3.141592653589793 6 5 01 Apr 2025 Counter({'a': 2, 'b': 1})
5 4 3.0 2.7183 10
```

Style	When
<code>import pandas as pd</code>	Standard aliases (<code>np</code> , <code>pd</code> , <code>plt</code> , <code>sns</code>)
<code>from pathlib import Path</code>	A few names used often
<code>from module import *</code>	Avoid: hides where names come from

Writing your own module

Python

```
from pathlib import Path
Path("metrics.py").write_text('"""Reusable business metrics."""')
GST = 0.18

def margin_pct(profit, sales):
    return round(100 * profit / sales, 1) if sales else None

def with_gst(amount):
    return round(amount * (1 + GST), 2)

if __name__ == "__main__":
    # runs only when executed directly, not when imported
    print("self-test:", margin_pct(25, 200))
    '''', encoding="utf-8")
import importlib, sys
if "" not in sys.path: sys.path.insert(0, "") # notebooks and scripts already search
the current folder
import metrics
importlib.reload(metrics)
print(metrics.margin_pct(30, 120), metrics.with_gst(1000), metrics.__doc__)
import subprocess, sys
print(subprocess.run([sys.executable, "metrics.py"], capture_output=True,
text=True).stdout.strip())
Path("metrics.py").unlink()
```

OUTPUT

```
25.0 1180.0 Reusable business metrics.
self-test: 12.5
```

`__name__` is `"__main__"` when the file is run as a script and the module name when imported. The guard lets one file be both a library and a script.

Where does Python look?

The current folder, then the folders in `sys.path` (including `site-packages` , where pip installs libraries). `ModuleNotFoundError` usually means: wrong environment, package not installed, or a

typo. Never name your own file `pandas.py` or `random.py` : it will shadow the real library.

Packages and project layout

A **package** is a folder of modules (with an `__init__.py`). A clean data project:

Text

```
churn-project/
├── pyproject.toml           # project metadata and dependencies
├── README.md
├── .gitignore              # venv, data, secrets, __pycache__
├── data/                   # raw/ and processed/ (usually not committed)
├── notebooks/              # exploration
├── src/churn/
│   ├── __init__.py
│   ├── load.py             # read and validate data
│   ├── features.py         # feature engineering
│   └── train.py            # model training
└── tests/
    └── test_features.py
```

Import with `from churn.features import add_tenure_band` . Notebooks for exploring; `.py` modules for anything reused or scheduled.

The standard library tour

Python

```
import os, random, statistics, datetime as dt, itertools, textwrap, uuid, time
random.seed(42)
print(random.randint(1, 6), random.choice(["N", "S", "E"]), random.sample(range(100), 3),
round(random.random(), 3))
print(statistics.stdev([2, 4, 4, 4, 5, 5, 7, 9]), statistics.mode([1, 2, 2, 3]))
today = dt.date(2025, 10, 27); due = today + dt.timedelta(days=45)
print(due, due.strftime("%A"), (dt.date(2026, 1, 1) - today).days, "days to new year")
print(os.cpu_count() > 0, os.getenv("HOME") is not None, textwrap.shorten("A very long
product description here", 20), len(str(uuid.uuid4())))
```

OUTPUT

```
6 N [3, 94, 35] 0.245
2.138089935299395 2
2025-12-11 Thursday 66 days to new year
True True A very long [...] 36
```

Module	Use
<code>pathlib</code> , <code>os</code> , <code>shutil</code> , <code>glob</code>	Files and folders
<code>datetime</code> , <code>zoneinfo</code> , <code>time</code>	Dates, time zones, timing
<code>math</code> , <code>statistics</code> , <code>random</code> , <code>decimal</code>	Numbers (<code>decimal</code> for money)
<code>collections</code> , <code>itertools</code> , <code>functools</code>	Data structures and iteration helpers
<code>csv</code> , <code>json</code> , <code>sqlite3</code> , <code>pickle</code>	Data formats and storage
<code>re</code>	Regular expressions
<code>logging</code> , <code>argparse</code> , <code>subprocess</code>	Scripts and tools
<code>urllib</code> , <code>http</code>	Web (most people use <code>requests</code> / <code>httpx</code>)
<code>concurrent.futures</code> , <code>asyncio</code>	Parallel and asynchronous work

pip, virtual environments and uv

Each project should have its **own virtual environment** so its library versions do not clash with other projects.

Text

```
# Create and activate an environment (built-in venv)
python -m venv .venv
source .venv/bin/activate          # macOS/Linux
.venv\Scripts\activate            # Windows

# Install, list, pin, reproduce
pip install pandas scikit-learn matplotlib
pip list
pip freeze > requirements.txt
pip install -r requirements.txt

# uv: a much faster modern alternative (by Astral)
uv init churn-project && cd churn-project
uv add pandas scikit-learn
uv run python src/churn/train.py
```

Virtual environments



The project isolation model: each project gets its own environment and its own library versions.

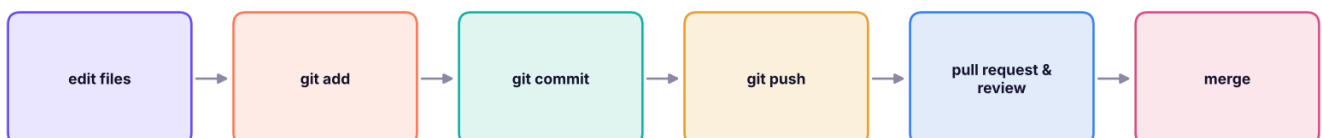
TIP Conda, Anaconda, Colab

Anaconda/Miniconda bundle Python with data libraries and the `conda` environment manager; popular in data science courses. **Google Colab** and **Kaggle Notebooks** run Python in the browser with libraries pre-installed: ideal for practice. Pick one tool per project and stick to it.

Git and GitHub essentials

Git tracks changes to files; **GitHub** hosts repositories online for backup, collaboration and your portfolio.

Everyday Git workflow



The everyday Git cycle.

Text

```
git init                # start tracking a folder
git status              # what changed?
git add analysis.py     # stage a file (git add . = everything)
git commit -m "Add churn EDA" # save a snapshot with a message
git log --oneline        # history
git switch -c feature/rfm # new branch
git push -u origin feature/rfm # upload the branch to GitHub
git pull                # download others' changes
git clone https://github.com/user/repo.git
git diff                # unstaged changes
git restore file.py     # discard local changes to a file
```

A `.gitignore` keeps junk and secrets out:

Text

```
.venv/
__pycache__/
.ipynb_checkpoints/
data/raw/
*.env
.DS_Store
```

WATCH OUT Never commit secrets

API keys, passwords and tokens belong in environment variables or a `.env` file listed in `.gitignore`. Read them with `os.getenv("OPENAI_API_KEY")`. If a key is ever pushed, revoke it immediately: deleting the commit is not enough.

Summary

KEY TAKEAWAYS

- A module is a `.py` file; a package is a folder of modules. Use standard aliases and avoid `import *`.
- `if __name__ == "__main__":` separates script code from importable code.
- One virtual environment per project; pin versions in `requirements.txt` or `pyproject.toml` (pip, uv or conda).
- The standard library covers files, dates, maths, data formats, logging and more.
- Git: add → commit → push → pull request → merge; use `.gitignore` and never commit secrets.

Quick quiz

6 questions · tap an answer to check it

1. What does `if __name__ == "__main__":` do?

A Imports main

B Runs the block only when the file is executed directly

C Defines the main function

D Speeds up the program

Answer: B. When imported, `__name__` is the module name.

2. Why use a virtual environment?

- A It makes Python faster
- B It isolates each project's library versions
- C It is required by Git
- D It encrypts the code

Answer: B. Different projects may need different versions.

3. How do you record the exact installed versions for others?

- A `pip list`
- B `pip freeze > requirements.txt`
- C `git add .`
- D `pip update`

Answer: B. Others recreate with `pip install -r requirements.txt`.

4. Why should you not name your file `pandas.py` ?

- A It is too long
- B It shadows the real pandas library on import
- C Git rejects it
- D It is reserved

Answer: B. Python finds your file first.

5. Which Git command uploads your commits to GitHub?

A `git commit`

B `git push`

C `git add`

D `git log`

Answer: B. Commit saves locally; push uploads.

6. Where should API keys go?

A In the code

B In environment variables or an ignored `.env` file

C In the README

D In a commit message

Answer: B. Never commit secrets.

EXERCISE Setup checklist

(1) Create a project folder with a virtual environment and install pandas and matplotlib. (2) Write a module `cleaning.py` with two functions and import it from a notebook. (3) Create a GitHub repository, add a `.gitignore` and a README, and push. (4) Create a branch, change the README and open a pull request to yourself.

Show solution (worked answer)

Follow the commands in this lesson. Check with `python -c "import pandas; print(pandas.__version__)"` and `git log --oneline`. A good README states the question, the data source, how to run the code and the key findings with one chart.

INTERVIEW CORNER

Questions you may be asked

Q: Difference between a module, a package and a library? A module is one file, a package is a folder of modules, and a library is a collection of packages distributed together (e.g. on PyPI).

Q: How do you make a Python project reproducible? Virtual environment, pinned dependencies, a README with run instructions, fixed random seeds, version control, and data lineage.

Q: git merge vs git rebase? Merge combines histories with a merge commit; rebase replays your commits on top of another branch for a linear history. Never rebase shared branches.

Practise more online

Free, trusted places to test yourself on this topic. Links open the original site in a new tab.