**1. Comprehensions vs map/filter**

**Comprehensions:**

* Pythonic, readable way to transform/filter data.
* Example:

python

squares = [x\*\*2 for x in range(5)]

evens = [x for x in range(10) if x % 2 == 0]

**map() and filter():**

* Functional style, uses functions instead of inline expressions.
* Example:

python

squares = list(map(lambda x: x\*\*2, range(5)))

evens = list(filter(lambda x: x % 2 == 0, range(10)))

**Key Difference:**  
Comprehensions are usually more readable and Pythonic.  
map/filter may be more concise when reusing existing functions.

**2. Decorator Primer**

A **decorator** is a function that wraps another function to add extra behavior without modifying the original code.

Example:

python

def log\_time(func):

def wrapper(\*args, \*\*kwargs):

print("Starting function...")

result = func(\*args, \*\*kwargs)

print("Function ended.")

return result

return wrapper

@log\_time

def process():

print("Processing...")

process()

Use cases: logging, timing, caching, authorization, retry logic, etc.

**3. venv vs poetry**

**venv:**

* Built-in virtual environment tool.
* Manages isolated Python environments.
* Requires pip and requirements.txt.

**poetry:**

* Modern tool for **dependency management** + packaging.
* Handles virtual environments, lock files, build, publish.
* Uses pyproject.toml.

**Comparison:**

| **Feature** | **venv** | **poetry** |
| --- | --- | --- |
| Virtualenv | Yes | Yes |
| Dependency Resolver | Manual | Automatic |
| Packaging | Manual (setuptools) | Built-in |
| Lock file | No | Yes (poetry.lock) |
| CLI | Simple | Rich CLI |

**4. Jupyter Magics**

Magics are special Jupyter Notebook commands that start with % (line magics) or %% (cell magics).

* %time: Time a single line

python

%time sum(range(1000000))

* %%time: Time the whole cell

python

%%time

total = 0

for i in range(1000000):

total += i

* %debug: Opens the interactive debugger after an exception.

**5. PyTest Fixtures & Parametrization**

**Fixtures:**

* Provide reusable setup for tests (e.g., test data, config).

python

import pytest

@pytest.fixture

def sample\_data():

return [1, 2, 3]

def test\_sum(sample\_data):

assert sum(sample\_data) == 6

**Parametrization:**

* Run same test with multiple inputs.

python

@pytest.mark.parametrize("x,expected", [(2, 4), (3, 9)])

def test\_square(x, expected):

assert x\*\*2 == expected

**6. Coverage Report**

Measures how much of your code is executed by tests.

Run with:

bash

coverage run -m pytest

coverage report

coverage html # Opens in browser

Goal: >80% is typically considered good for reliability.

**7. Build Wheel & Publish to Private PyPI**

**Build:**

bash

pip install build

python -m build

Creates .whl and .tar.gz files in dist/

**Publish to private PyPI:**

Use tools like twine:

bash

pip install twine

twine upload --repository-url https://your.private.repo dist/\*

Add credentials in ~/.pypirc.

**8. Git Pre-commit Hook**

Runs checks before every git commit to ensure quality.

Steps:

1. Install pre-commit:

bash

pip install pre-commit

1. Create .pre-commit-config.yaml:

yaml

repos:

- repo: https://github.com/psf/black

rev: 24.3.0

hooks:

- id: black

- repo: https://github.com/pycqa/flake8

rev: 7.0.0

hooks:

- id: flake8

1. Activate:

bash

pre-commit install

Now, on every commit, it will auto-format code, check for issues, and fail the commit if anything is wrong.