Best Practices in python™ Programming

Thomas Isensee, Scientific Software Center (SSC), Heidelberg University

Course slides

Last updated: 2026-09-07

Outline

Scientific Software Center

1 Course Overview

Course overview

Course focus

Python is widely used in science and research, from quick data analysis scripts to reusable packages and simulation workflows.

This course focuses on practical habits and tools that make Python code easier to read, test, review, and share with collaborators.

What it's not

This is not a full introduction to:

- Git
- type checking
- testing
- packaging

We cover the minimum useful practices and point to dedicated SSC courses.

Related SSC course material

- Python testing: Introduction to Python Testing
- Python packaging: Python Packaging
- Project template: SSC Python project template
- Project cookiecutter: SSC Python package template

Scientific Software Center

2 Project Setup

Questions a project should answer

Scientific Software Center

Before talking about code style, a project should make clear:

- Which Python environment should I use?
- Where are dependencies recorded?
- Is this a script or a package?
- Where are tests expected to live?
- Is the project under version control?

Git basics for this course

Scientific Software Center

Git is the history and collaboration layer for this development workflow.

Most important aspects:

- clone a repository with `git clone <repo-url>`
- create or edit files
- inspect changes with `git status` and `git diff`
- commit changes with `git commit`
- push committed changes to the remote repository with with `git push`
- `.gitignore` for files that should not be committed

For this course:

- `gh student accept ssciwr-courses pbp-2026-09-16 <assignment-name>`
- `gh student submit`

Why isolated environments?

An isolated environment keeps a project's dependencies separate from your system Python and from other projects.

Benefits:

- different projects can need different versions
- installed tools do not leak between projects
- collaborators can recreate the setup
- CI and local development can use the same dependency list

venv

Scientific Software Center

`venv` is part of Python and is a good default when you only need Python packages from PyPI.

```
python -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install ruff pytest mypy
```

Add `.venv/` to `.gitignore`.

Conda

Scientific Software Center

Conda is common in scientific Python, especially when a project depends on compiled libraries or non-Python packages.

```
conda create -n my-project python=3.14.7
conda activate my-project
conda install numpy scipy
python -m pip install ruff pytest
```

uv is a newer Python project and package manager written in Rust.

It can:

- create environments
- install dependencies
- run tools
- manage lock files
- support workspaces

Highly recommended, not required for this course.

Scientific Software Center

3 PEPs, Style, Ruff, and pre-commit

Scientific Software Center

PEP 8

Python Enhancement Proposals (PEPs) describe Python language changes, processes, and conventions.

- PEP index: peps.python.org
- PEP 8: style guide for Python code

PEP 8 covers naming, indentation, imports, whitespace, comments, and layout conventions.

For this course, the most important message is:

- prefer consistency over personal taste
- prefer readable code over clever code
- automate formatting and linting whenever possible
- adapt when a project already has a local style

- Use 4 spaces per indentation level.
- Put imports at the top of the file.
- Prefer one import per line.
- Use `lowercase_with_underscores` for functions and variables.
- Use `CamelCase` for classes.
- Avoid ambiguous names such as `l`, `O`, and `I`.
- Keep comments and code consistent.
- BORING!

Scientific Software Center

PEP 8

Python Enhancement Proposals (PEPs) describe Python language changes, processes, and conventions.

- PEP index: peps.python.org
- PEP 8: style guide for Python code

PEP 8 covers naming, indentation, imports, whitespace, comments, and layout conventions.

For this course, the most important message is:

- prefer consistency over personal taste
- prefer readable code over clever code
- automate formatting and linting whenever possible
- adapt when a project already has a local style

Easter Egg: Type `import this` and find enlightenment.

- Use 4 spaces per indentation level.
- Put imports at the top of the file.
- Prefer one import per line.
- Use `lowercase_with_underscores` for functions and variables.
- Use `CamelCase` for classes.
- Avoid ambiguous names such as `l`, `O`, and `I`.
- Keep comments and code consistent.
- BORING!

Scientific Software Center

Easter Egg: Type `import this` and find enlightenment.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Automated style feedback

Scientific Software Center

Manual style review is expensive and not very interesting (it’s boring!).

Let tools handle the boring parts so humans can review:

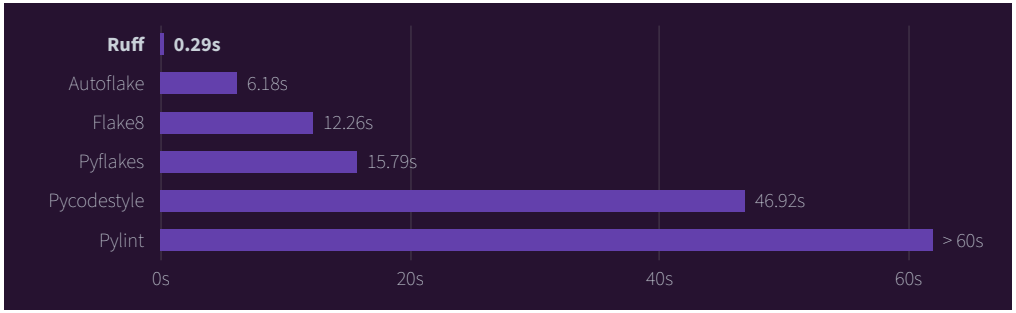
- names
- structure / architecture
- tests
- scientific correctness
- performance

Scientific Software Center

Ruff is a fast Python linter and formatter.

In this course we use Ruff instead of teaching separate tools such as:

- Flake8
- isort
- Black
- pyupgrade



It's very fast!

Install and run Ruff

Scientific Software Center

Install Ruff via `python -m pip install ruff`

Linting

Linters focus on code health and bug detection.

```
ruff check path/to/code.py
ruff check path/to/package/
```

or

```
# Apply safe automatic lint fixes
ruff check --fix .
```

or check Jupyter notebooks

```
ruff check analysis.ipynb
```

Formatting

Formatters focus on visual style and rewrite text layout.

```
ruff format path/to/code.py
ruff format path/to/package/
```

or

```
# Check formatting without changing files
ruff format --check .
```

Reading Ruff messages

Scientific Software Center

Ruff reports the rule, location, code context, and often a fix.

```
F401 [*] `os` imported but unused
 --> sensor_report.py:6:8
  |
4 | import statistics
5 | from pathlib import Path
6 | import os
  |        ^^
7 |
8 | DATA_FILE=Path(__file__).parent / "data" / "readings.csv"
  |
help: Remove unused import: `os`
  |
5 | from pathlib import Path
  - import os
```

- `F401` is the rule code.
- `sensor_report.py:6:8` is file, line, and column.
- `[*]` means Ruff can fix this automatically.
- the `help` line explains the change.

Configuring Ruff

Scientific Software Center

`pyproject.toml`:

```toml
[tool.ruff]
line-length = 79

[tool.ruff.lint]
select = [
    "E",  # pycodestyle errors
    "F",  # Pyflakes
    "B",  # flake8-bugbear
    "I",  # import sorting
    "UP", # pyupgrade
    "SIM",
]
```

`ruff.toml`

```toml
line-length = 79

[lint]
select = ["E", "F", "B", "I", "UP", "SIM"]
```

Start with a small rule set and add more rules deliberately.

pre-commit and Ruff

Scientific Software Center

`pre-commit` runs checks before a commit is created.

Install/Setup

This helps when a project starts clean, then collaborators join and not everyone uses the same editor setup.

```
python -m pip install pre-commit
pre-commit install
```

Configuration

`.pre-commit-config.yaml` in repository root:

```yaml
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.16.6
    hooks:
      - id: ruff-check
        args: [--fix, --show-fixes]
      - id: ruff-format
```

Run all hooks manually with `pre-commit run --all-files`.

pre-commit and other tools

Scientific Software Center

`pre-commit` can apply other useful tools as well.

General

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v6.0.0
    hooks:
      - id: end-of-file-fixer
      - id: check-added-large-files
      - id: check-json
      - id: check-toml
      - id: check-yaml
      - id: mixed-line-ending
      - id: requirements-txt-fixer
      - id: trailing-whitespace
```

Always good to have.

nbstripout

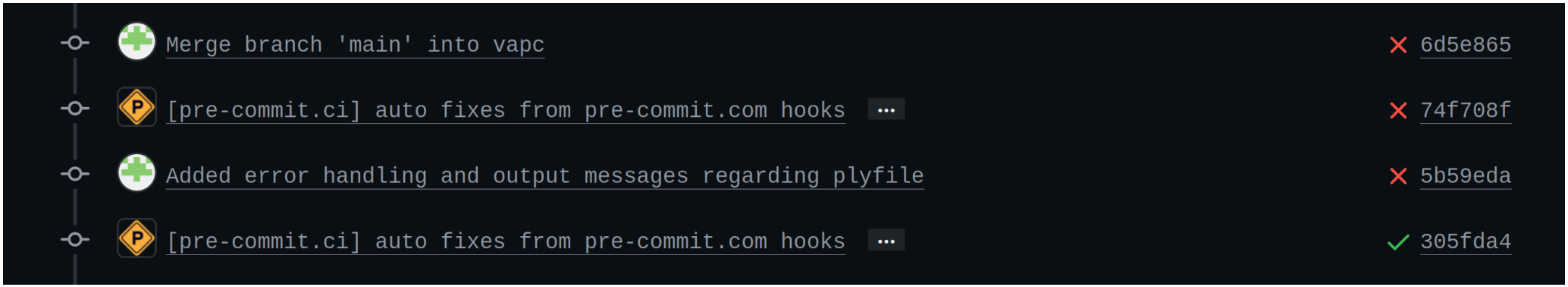
```yaml
repos:
  - repo: https://github.com/kynan/nbstripout
    rev: 0.9.1
    hooks:
      - id: nbstripout
        files: ".ipynb"
```

Remove output from Jupyter notebooks. Commit only code.

pre-commit and GitHub

Scientific Software Center

Services such as `pre-commit.ci` can run the same hooks on pull requests and even commit automatic fixes.



Your colleagues won’t be able to mess up your code!

Task 1: `gh student accept ssciwr-courses pbp-2026-09-16 ruff`

Scientific Software Center

1. Install Ruff and pre-commit via `python -m pip install ruff pre-commit`
2. Run `ruff check .` and inspect the output.
3. Run `ruff check --fix .` and inspect the diff via `git diff`.
4. Run `ruff format .` and inspect the diff via `git diff`.
5. Add a pre-commit configuration file `.pre-commit-config.yaml`.

`.pre-commit-config.yaml` in repository root:

```yaml
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.16.6
  hooks:
    - id: ruff-check
      args: [--fix, --show-fixes]
    - id: ruff-format
```

6. Run `pre-commit install`.
7. `git add <changed-files>`.
8. Commit changes via `git commit -m "Fix format"`
9. Run `gh student submit` for submitting.

Scientific Software Center

4 Python packaging

Script or package?

Scientific Software Center

Not every file needs to become a package. A script with a small test function can stay a script, or a notebook.

Consider a package when code is:

- imported from multiple scripts or notebooks
- reused by collaborators
- tested in CI
- documented as an importable API
- installed into an environment
- published or archived as a reusable tool

Scientific Software Center

Project structure

```
my-project/
├── README.md
├── pyproject.toml
├── src/
|   └── my_project/
|       ├── __init__.py
|       └── analysis.py
└── tests/
    └── test_analysis.py
```

`src/` helps tests use the installed package.

`tests/` keeps tests separate from library code.

`README.md` contains general information about the repository and is displayed as front page on GitHub and others.

Minimal package-shaped project

Project structure

```
my-project/
├── README.md
├── pyproject.toml
├── src/
|   └── my_project/
|       ├── __init__.py
|       └── analysis.py
└── tests/
    └── test_analysis.py
```

`src/` helps tests use the installed package.

`tests/` keeps tests separate from library code.

`README.md` contains general information about the repository and is displayed as front page on GitHub and others.

Project metadata

`pyproject.toml` started as build-system configuration, but today it is also the standard place for project metadata and tool configuration.

It can record:

- project metadata (Version, description, authors etc.)
- supported Python versions
- dependencies
- optional development dependencies
- build-backend and tool settings for Hatchling, Ruff, pytest, mypy, and others

Scientific Software Center

File content

```toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "my-analysis"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = ["numpy", "pandas"]

[project.optional-dependencies]
dev = ["pytest", "ruff", "mypy"]
```

The key idea: another person or CI should be able to recreate the setup without any effort.

Project metadata

`pyproject.toml` started as build-system configuration, but today it is also the standard place for project metadata and tool configuration.

It can record:

- project metadata (Version, description, authors etc.)
- supported Python versions
- dependencies
- optional development dependencies
- build-backend and tool settings for Hatchling, Ruff, pytest, mypy, and others

Tool configuration example

Scientific Software Center

File content

```toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "my-analysis"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = ["numpy", "pandas"]

[project.optional-dependencies]
dev = ["pytest", "ruff", "mypy"]
```

The key idea: another person or CI should be able to recreate the setup without any effort.

```toml
[tool.ruff]
line-length = 79
```

```toml
[tool.pytest.ini_options]
testpaths = ["tests"]
```

Project metadata

`pyproject.toml` started as build-system configuration, but today it is also the standard place for project metadata and tool configuration.

It can record:

- project metadata (Version, description, authors etc.)
- supported Python versions
- dependencies
- optional development dependencies
- build-backend and tool settings for Hatchling, Ruff, pytest, mypy, and others

Task 2: `gh student accept ssciwr-courses pbp-2026-09-16 packaging`

Scientific Software Center

1. Install pytest via `python -m pip install -r requirements-dev.txt`.
2. Run `python example.py` and `python -m pytest`.
3. Create `src/measurement_tools/` and move `measurement_tools.py` to `analysis.py` there.
4. Add `__init__.py` and the minimal `pyproject.toml` shown in the README.
5. Install via `python -m pip install -e .`.
6. Run the example and tests again.
7. Commit and submit via `gh student submit`.

```
measurement-tools/
├── .gitignore
├── LICENSE.md
├── pyproject.toml
├── README.md
├── src/
│   └── measurement_tools/
│       ├── __init__.py
│       └── analysis.py
└── tests/
    └── test_measurement_tools.py
```

`__init__.py`:

```python
from .analysis import center_measurements, mean_measurement

__all__ = ["center_measurements", "mean_measurement"]
```

Run cookiecutter, answer questions

```
$ cookiecutter cookiecutter-python-package/
  [1/10] project_name (My Project): my-project
  [2/10] remote_url (None):
  [3/10] project_slug (my-project):
  [4/10] full_name (Your Name): My Name
  [5/10] Select license
    1 - MIT
    2 - BSD-2
    3 - GPL-3.0
    4 - LGPL-3.0
    5 - None
    Choose from [1/2/3/4/5] (1):
  [6/10] Select github_actions_ci
    1 - Yes
    2 - No
    Choose from [1/2] (1):
    .
    .
```

Insepect new package

```
├── CITATION.cff
├── codecov.yml
├── COPYING.md
├── doc/
├── FILESTRUCTURE.md
├── LICENSE.md
├── notebooks
│   └── demo.ipynb
├── pyproject.toml
├── README.md
├── src
│   └── my_project
│       └── __init__.py
├── tests
│   ├── conftest.py
│   ├── __init__.py
│   └──
test_my_project.py
└── TODO.md
```

Follow instructions in `TODO.md` to finish the setup of your new package.

Scientific Software Center

5 Tests and Type Hints

Why tests are essential?

Scientific Software Center

Tests are executable assumptions

Tests are executable assumptions. They help answer:

- Does the code do what I think it does?
- Did my recent change break old behavior?
- Can a collaborator change this without guessing?
- Does the code still work on a clean machine or in CI?

Tests in scientific code

- For scientific code, tests are not only software engineering hygiene.
- They are part of making computational results trustworthy.
- This course only gives a short introduction.

Full course: Introduction to Python Testing

A tiny pytest example

Scientific Software Center

Code under test

```
def mean(values):
    return sum(values) / len(values)
```

```
python -m pytest
```

A pytest example

```
import pytest

from statistics_tools import mean

def test_mean_of_three_values():
    assert mean([1.0, 2.0, 3.0]) == 2.0

def test_mean_of_empty_list_raises():
    with pytest.raises(ZeroDivisionError):
        mean([])
```

The pattern:

- put expected behavior in code
- cover normal cases and edge cases
- run tests before and after refactoring
- ideally run tests in GitHub CI (or other) upon any pull request

Type hints

Scientific Software Center

Type hints describe what kind of values code expects and returns:

```
def total(values: list[float]) -> float:
    return sum(values)
```

Python does not normally enforce these annotations at runtime. Their main value is that they help readers, editors, and type-checking tools.

Type hints: `None` and defaults

Scientific Software Center

A type annotation and a default answer different questions:

Which values are allowed?

```
def report(label: str | None): ...
```

The caller must provide `label`, but its value may be a string or `None`: `report(None)` is valid; `report()` is not.

May the argument be omitted?

```
def report(label: str = "Results"): ...
```

The caller may omit `label`. Its default is `"Results"`, and its declared type does not include `None`.

Both ideas can be combined:

```
def report(
    values: list[float],
    label: str | None = None,
    precision: int = 2,
) -> str:
    ...
```

`label` may be omitted or set to `None`; `precision` may be omitted but should be an `int` when supplied.

Installation/Setup

Install mypy:

```
python -m pip install mypy
```

Run it:

```
mypy path/to/code.py
```

example.py

```
def total(values: list[float]) -> float:
    return sum(values)

measurements = ["1.0", "2.0", "3.0"]
print(total(measurements))
```

```
14:00 $ mypy example.py
example.py:5: error: Argument 1 to "total" has incompatible type
"list[str]"; expected "list[float]"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

At runtime this fails. A type checker can flag the mistake earlier: `list[str]` is not `list[float]`.

Task 3: `gh student accept ssciwr-courses pbp-2026-09-16 tests`

Scientific Software Center

1. Install mypy via `python -m pip install mypy`.
2. Run `python -m sample_summary` and `python -m pytest`.
3. Now run `python -m mypy sample_summary.py`.
4. Examine the issues and fix them.

```
sample_summary.py:18: error: List comprehension has incompatible type List[str]; expected List[float]  [misc]
sample_summary.py:31: error: Function is missing a type annotation  [no-untyped-def]
sample_summary.py:41: error: Call to untyped function "render_summary" in typed context  [no-untyped-call]
Found 3 errors in 1 file (checked 2 source files)
```

Task 3: `gh student accept ssciwr-courses pbp-2026-09-16 tests`

Scientific Software Center

1. Install mypy via `python -m pip install mypy`.
2. Run `python -m sample_summary.py` and `python -m pytest`.
3. Now run `python -m mypy sample_summary.py`.
4. Examine the issues and fix them.

```
    return [float(row[1]) for row in rows]
```

```
def render_summary(total: float, accepted: bool) -> str:
```

5. Extend `tests/test_sample_summary.py` with a test `test_tolerance_boundary_is_included` expressing this requirement:

```
A measured value exactly one tolerance away from its target is accepted. For example, `9.5` is within a tolerance of
`0.5` around `10.0`.
```

Task 3: `gh student accept ssciwr-courses pbp-2026-09-16 tests`

Scientific Software Center

1. Install mypy via `python -m pip install mypy`.
2. Run `python -m sample_summary.py` and `python -m pytest`.
3. Now run `python -m mypy sample_summary.py`.
4. Examine the issues and fix them.

```
    return [float(row[1]) for row in rows]
```

```
def render_summary(total: float, accepted: bool) -> str:
```

5. Extend `tests/test_sample_summary.py` with a test `test_tolerance_boundary_is_included` expressing this requirement:

```
A measured value exactly one tolerance away from its target is accepted. For example, `9.5` is within a tolerance of
`0.5` around `10.0`.
```

```
def test_tolerance_boundary_is_included() -> None:
    assert is_within_tolerance(9.5, target=10.0, tolerance=0.5)
```

6. Run `python -m pytest` again.
7. Run `gh student submit` for submitting.

Scientific Software Center

6 Common Python Pitfalls

Names refer to objects

Scientific Software Center

Assigning a list to another name does not copy it:

```
original = [[1, 2], [3, 4]]
alias = original
alias[0].append(99)
print(original)  # [[1, 2, 99], [3, 4]]
```

Both names refer to the same object. A mutation through either name is visible through the other.

Copy nested data deliberately

A shallow copy creates a new outer list, but nested objects are still shared:

```
copied = original.copy()
copied[0].append(99)
print(original)  # [[1, 2, 99], [3, 4]]
```

When nested values must be independent, copy the known structure explicitly or use `deepcopy` deliberately:

```
from copy import deepcopy

copied = deepcopy(original)
```

`deepcopy` may copy more than intended. Copy only when independent state is actually required.

Mutable default arguments

Scientific Software Center

Defaults are evaluated once, so this list is shared between calls:

```python
def add_sample(value: float, samples: list[float] = []) -> list[float]:
    samples.append(value)
    return samples
```

```
>>> add_sample(0.0)
[0.0]
>>> add_sample(1.0)
[0.0, 1.0]
>>> add_sample(2.0)
[0.0, 1.0, 2.0]
```

Test this by calling the function twice.

Use `None` and create a new list inside the function:

```python
def add_sample(value: float, samples: list[float] | None = None) -> list[float]:
    samples = [] if samples is None else samples
    samples.append(value)
    return samples
```

```
>>> add_sample(0.0)
[0.0]
>>> add_sample(1.0)
[1.0]
>>> add_sample(2.0)
[2.0]
```

Compare floating-point values deliberately

```
total = 0.1 + 0.2
total == 0.3  # False
```

Use a tolerance appropriate for the domain:

```
import math
import pytest

math.isclose(total, 0.3, rel_tol=1e-9)
assert total == pytest.approx(0.3, rel=1e-9)
```

Boundary cases still need tests; type checking cannot find a wrong `<` where `<=` was intended.

Do not silence failures

Scientific Software Center

A broad exception handler can silently turn bad input into plausible output:

```
try:
    return float(text)
except Exception:
    return 0.0
```

Catch only errors you can handle, and retain useful context:

```
try:
    return float(text)
except ValueError as error:
    raise ValueError(f"Invalid measurement: {text!r}") from error
```

Missing is not the same as falsy

Scientific Software Center

`0`, `0.0`, empty containers, and `None` are all falsy, but they do not necessarily mean the same thing:

```
def effective_tolerance(tolerance: float | None) -> float:
    if not tolerance:
        return 0.1  # Also replaces a valid 0.0.
    return tolerance
```

If only `None` means “not provided”, test for it explicitly:

```
if tolerance is None:
    return 0.1
```

Assertions are not input validation

Scientific Software Center

Assertions are useful in tests and for developer assumptions, but Python can remove them when run with optimization:

```
assert tolerance >= 0
```

Validate external input with an explicit exception:

```
if tolerance < 0:
    raise ValueError("tolerance must be non-negative")
```

Imports can find the wrong file

Scientific Software Center

Why does this fail?

```
project/
├── analysis.py
└── statistics.py
```

`analysis.py`:

```python
import statistics

values = [1.0, 2.0, 3.0]
print(statistics.mean(values))
```

What did Python import?

```
AttributeError: module 'statistics'
has no attribute 'mean'
```

Inspect the imported module:

```python
print(statistics.__file__)
```

```
.../project/statistics.py
```

Python found the local file instead of the standard-library module.

Rename it to something project-specific, such as `sample_statistics.py`.

Import-time side effects

Module-level code runs during import. Keep reusable logic in functions and command-line execution behind the `__name__` guard:

```python
def main() -> None:
    print("Running analysis")

if __name__ == "__main__":
    main()
```

The guard prevents `main()` from running when tests or other programs import the module.

Task 4: `gh student accept ssciwr-courses pbp-2026-09-16 pitfalls`

Scientific Software Center

1. Add a test that checks record_warning with default argument.

```python
def test_record_warning_uses_a_fresh_default() -> None:
    assert record_warning("first") == ["first"]
    assert record_warning("second") == ["second"]
```

2. Fix the mutable default argument.

```python
def record_warning(
    message: str,
    warnings: list[str] | None = None,
) -> list[str]:
    if warnings is None:
        warnings = []
    warnings.append(message)
    return warnings
```

3. Add test for sum of two measurements.

```python
def
test_total_value_uses_approximate_float_comparison() ->
None:
    measurements = [
        Measurement("A", 0.1, "mm"),
        Measurement("B", 0.2, "mm"),
    ]

    assert total_value(measurements) ==
pytest.approx(0.3)
```

4. Test that zero is a valid tolerance.

```python
def test_zero_is_a_valid_tolerance() -> None:
  assert effective_tolerance(0.0) == 0.0
```

Task 4: `gh student accept ssciwr-courses pbp-2026-09-16 pitfalls`

Scientific Software Center

5. Fix `effective_tolerance`.

```
def effective_tolerance(requested: float | None) ->
float:
    """Return the requested tolerance or the workflow
default."""
    if requested is None:
        return DEFAULT_TOLERANCE
    return requested
```

6. Test that invalid measurements raise an error.

```
  def test_parse_invalid_measurement() -> None:
      with pytest.raises(ValueError, match="not-a-
number"):
          parse_measurement("A5", "not-a-number", "C")
```

7. Let possible errors propagate and do not invent replacement values.

```
def parse_measurement(sample_id: str, text: str, unit:
str) -> Measurement:
    """Parse one measurement exported by an
instrument."""
    value = float(text)
    return Measurement(sample_id, value, unit)
```

8. Run `python -m pytest` again.
9. Run `ruff check --fix .` and `ruff format .`
10. Run `gh student submit` for submitting.

Scientific Software Center

7 Practical Python Patterns

Python provides small, composable tools that make programs easier to understand, use correctly, and maintain.

This section covers patterns for:

- communicating intent clearly
- modeling structured data explicitly
- handling paths and resources safely
- making runtime behavior observable
- expressing common iterations directly

Docstrings

Scientific Software Center

✗

```
def some_function(par1: str, par2: str) -> str:
    return par1 + "/" + par2 + ".txt"
```

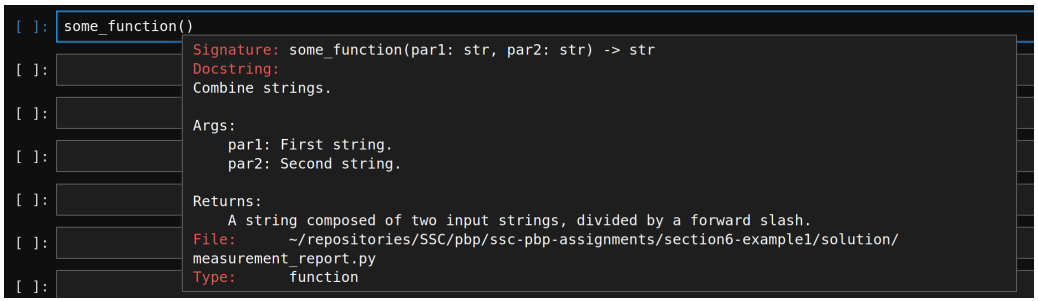
✓

```
def some_function(par1: str, par2: str) -> str:
    """Combine strings.

    Args:
        par1: First string.
        par2: Second string.

    Returns:
        A string composed of two input strings, divided by a
forward slash.
    """
    return par1 + "/" + par2 + ".txt"
```

Shift + Tab shows useful info:



f-strings

Scientific Software Center

Use f-strings for readable string interpolation:

✗ (not wrong)

```
sample_id = "A5"
temperature = 21.378

message = "Sample " + sample_id + " measured " + str(temperature) + " C"
```

✓ (better)

```
sample_id = "A5"
temperature = 21.378

message = f"Sample {sample_id} measured {temperature:.1f} C"
```

The variable stays near the text that explains it.

pathlib

Scientific Software Center

`pathlib.Path` represents file system paths as objects instead of fragile strings.

String-based path handling ✗ (not wrong):

```
filename = data_dir + "/" + sample_id + ".csv"
```

Path-based handling ✓ (better):

```
from pathlib import Path

data_dir = Path("data")
filename = data_dir / f"{sample_id}.csv"
```

Uses appropriate paths depending on the OS (don't worry about "/" or "\" anymore.)

pathlib methods and benefits

Scientific Software Center

Benefits:

- clearer path joining with `/`
- fewer operating-system assumptions
- convenient methods for common file operations
- easier testing with temporary directories

```
path.exists()
path.is_file()
path.iterdir()
path.parent
path.name
path.suffix
path.stem
```

```
output_dir = Path("results")
output_dir.mkdir(parents=True,
exist_ok=True)
```

```
with open(path) as f:
    ...
```

```
text = path.read_text()
path.write_text("hello")
```

Relative paths depend on the working directory

`Path("data/results.csv")` is resolved relative to the current working directory, which may differ between a terminal, an editor, and CI.

Prefer passing paths into reusable functions. At the application boundary, choose the base directory explicitly:

```python
from pathlib import Path

def load_results(path: Path) -> str:
    return path.read_text(encoding="utf-8")

project_dir = Path(__file__).parent
load_results(project_dir / "data" / "results.csv")
```

Dataclasses make records explicit

Use a dataclass for a small record with a known set of named fields:

✗ (not wrong)

```
class Point:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point(x={self.x!r}, y={self.y!r})"

    def __eq__(self, other):
        if not isinstance(other, Point):
            return NotImplemented
        return self.x == other.x and self.y == other.y
```

✓ (better)

```
from dataclasses import dataclass

@dataclass(frozen=True, slots=True, order=True)
class Point:
    x: float
    y: float
```

Generates initializer, a readable representation, and value-based equality. The annotations document the fields, but do not validate values at runtime.

`frozen=True`: prevents reassignment of fields.
`slots=True`: generally reduces memory use.
`order=True`: generates <, <=, >, and >= (in field order).

Mutable fields need a factory

Scientific Software Center

Dataclasses reject a list as a direct default. Give each instance its own list with `default_factory`:

✗

```
@dataclass
class Mesh:
    vertices: list = []
```

```
ValueError: mutable default <class 'list'> for field
vertices is not allowed: use default_factory
```

This would conceptually risk all instances sharing the same list, so dataclasses reject it.

✓

```
@dataclass
class SamplingRun:
    values: list[float] = field(default_factory=list)
```

This is the dataclass equivalent of avoiding a mutable default argument. Factories also work for dictionaries, sets, and more complex defaults.

Dataclasses in scientific workflows

Dataclasses are useful when:

- each record has the same known set of fields
- values should travel with metadata such as units or sample identifiers
- functions pass the same record between processing stages
- equality and readable output make tests and debugging easier

It is especially suitable for configuration objects, coordinates, simulation parameters, records, and intermediate results.

They are often clearer than parallel lists or dictionaries with repeated string keys. They are not a replacement for large numerical arrays or tables; choose a container that matches the data and operations.

logging

Use `logging` for reusable code and `print` for small one-off scripts or final user-facing output.

✗

```
def load_samples(path):
    print("Loading samples from %s", path)
```

✓

```
import logging

logger = logging.getLogger(__name__)

def load_samples(path):
    logger.info("Loading samples from %s", path)
```

Every log message has a severity level:

```
logging.debug("Detailed diagnostic information")
logging.info("Program started")
logging.warning("Something looks suspicious")
logging.error("Something failed")
logging.critical("Something went very badly wrong")
```

At the application entry point, choose the desired level of the logger:

```
logging.basicConfig(level=logging.INFO)

logging.debug("debug")      # not shown
logging.info("info")        # shown
logging.warning("warning")  # shown
logging.error("error")      # shown
```

Context managers

Use context managers for resources that must be cleaned up:

✗

```
from pathlib import Path

path = Path("results.txt")

handle = path.open("w")
handle.write("done\n")
handle.close()
```

✓

```
from pathlib import Path

path = Path("results.txt")

with path.open("w") as handle:
    handle.write("done\n")
```

The file is closed even if an error occurs inside the block.

enumerate and zip

Scientific Software Center

enumerate

✗ (not wrong)

```
for index in range(len(values)):
    value = values[index]
    print(index, value)
```

✓ (better)

```
for index, value in enumerate(values):
    print(index, value)
```

zip

✗ (not wrong)

```
for index in range(len(sample_ids)):
    sample_id = sample_ids[index]
    temperature = temperatures[index]
    print(sample_id, temperature)
```

✓ (better)

```
for sample_id, temperature in zip(sample_ids, temperatures):
    print(sample_id, temperature)
```

Task 5: `gh student accept ssciwr-courses pbp-2026-09-16 readable-python`

Scientific Software Center

```python
@dataclass(frozen=True, slots=True)
class Measurement:
    """A measured value together with its scientific context."""

    sample_id: str
    value: float
    unit: str
```

```python
def report_path(output_dir: Path, sample_id: str) -> Path:
    """Return the path for one sample report.

    Returns:
        The path of the sample report.

    """
    return output_dir / f"{sample_id}.txt"
```

```python
def render_report(measurement: Measurement) ->
str:
    return (
        f"Sample {measurement.sample_id}:
{measurement.value:.2f} {measurement.unit}\n"
    )
```

```python
def save_report(path: Path, report: str) ->
None:
    path.parent.mkdir(parents=True,
exist_ok=True)
    with path.open("w", encoding="utf-8") as
handle:
        handle.write(report)
```

Scientific Software Center

8 Comment on AI/LLM usage

AI/LLM usage

1. Ask for best practices.
2. Ask for modern Python.
3. Ask for tests.
4. Ask for documentation comments.

Example `AGENTS.md`:

```
This is a collection of my analysis scripts. Please write modern Python (3.11 or higher), apply common best practice
techniques, and write reasonable tests in the test/ directory. Each (public) function should have a reasonable
documentation comment that explains at least the input parameters and the ouptut.

Run "ruff --check .", "ruff --format .", and "mypy ." to validate any changes.
```

5. Use a CLI agent (e.g., Codex), then you don’t have to copy-paste code into a chat.