Testing and Automation for Computational Science

AM215 - LECTURE 6

The AM215 Team

Today's Roadmap

▒▒ Part 1: Foundations of Software Testing

Why we test, and an introduction to the modern pytest framework.

▒▒ Part 2: Practical Testing for Scientific Code

Techniques for testing numerical algorithms, stochastic models, and complex systems.

▒▒ Part 3: From Testing to Automation (CI/CD)

The principles of Continuous Integration and an introduction to GitHub Actions.

▒▒ Part 4: GitHub Actions in Practice

Building automated workflows for testing and package publishing.

Part 1: Foundations of Software Testing

Motivation: Building Trust in Scientific Code

- **Correctness:** Does your code produce the right answer?
- **Robustness:** Does it handle edge cases and unexpected inputs?
- **Maintainability:** Can you refactor or add features without breaking existing functionality?

> **Lecture 3:** If results aren't reproducible, they can't be trusted.
>
> **Today:** If results aren't tested, how can they be trusted?

Testing is the engineering discipline we use to build and maintain trust in our computational work.

> **Q:** When should I start writing tests?
> **A:** Probably as soon are you are creating modules of classes and functions.

The "Built-in" Way: unittest

Python comes with a built-in testing framework, unittest. It's class-based and was inspired by Java's JUnit.

```python
import unittest
from my_model import physics_func

class TestMyModel(unittest.TestCase):
    def test_physics_func(self):
        # Special assertion methods
        self.assertEqual(physics_func(2), 4)
        self.assertAlmostEqual(physics_func(0.1), 0.01)

if __name__ == '__main__':
    unittest.main()
```

> This works, but it's verbose. The community has largely moved to a more "Pythonic" tool.

The Modern Standard: pytest

pytest is the de facto standard for testing in Python. It's simpler, more powerful, and more intuitive.

The same test in pytest:

```python
# tests/test_model.py
from my_model import physics_func
import pytest

def test_physics_func():
    # Just use a plain `assert`
    assert physics_func(2) == 4
    # Use pytest.approx for floating point
    assert physics_func(0.1) == pytest.approx(0.01)
```

To run:

```
pytest
```

> No classes, no special methods. Just assert. pytest does the rest.

Part 2: Practical Testing for Scientific Code with pytest

Test Structure and Discovery

`pytest` automatically finds and runs your tests if you follow simple conventions.

1. File and Function Naming:

- Test files must be named `test_*.py` or `*_test.py`.
- Test functions must be named `test_*()`.

2. Directory Structure (from Lecture 5):

```
my_project/
├── src/
│   └── my_model/
│       ├── __init__.py
│       └── physics.py
├── tests/
│   └── test_physics.py  # Tests for physics.py
└── pyproject.toml
```

> By placing tests in a `tests/` directory, you separate your test code from your library code, reinforcing the `src/` layout best practice from **Lecture 5**.

Configuring pytest

You can customize pytest's behavior using a configuration file to make test runs consistent and reproducible. The two most common files are pytest.ini and pyproject.toml.

Using pytest.ini**:** A simple, dedicated file.

```
[pytest]
testpaths = tests
addopts = -ra --cov=my_package
filterwarnings =
    error
    ignore::DeprecationWarning
```

Using pyproject.toml**:** Integrates with your project's main config file (from **Lecture 5**).

```
[tool.pytest.ini_options]
testpaths = ["tests"]
addopts = "-ra --cov=my_package"
filterwarnings = [
    "error",
    "ignore::DeprecationWarning",
]
```

> Both are valid. Using pyproject.toml keeps all your tool configuration in one place.

What is a "Unit Test"?

A **unit test** verifies a small, isolated piece of code—a "unit"—in your application.

- **What is a unit?** Typically a single function or method.
- **What does "isolated" mean?** The test should not depend on external systems like databases, networks, or complex dependencies. Its success or failure should depend only on the unit being tested.

> The goal is to test one thing at a time. This makes tests fast, reliable, and easy to debug when they fail.

Unit vs. Integration Tests

- **Unit Test:** Verifies a single, isolated piece of code (a function or method).
  - *Example:* `test_vector_addition()` checks only the `__add__` method.
  - *Goal:* Fast, precise feedback on individual components.
- **Integration Test:** Verifies that multiple components work together as a system.
  - *Example:* A test that runs a full simulation pipeline: loading data, running the model, and generating a result file.
  - *Goal:* Ensure the entire process is connected correctly.

> A good test suite has many fast unit tests and a smaller number of more comprehensive integration tests.

Organizing Tests with Markers

As your test suite grows, you'll want to run subsets of tests. `pytest` markers let you label tests to categorize them.

Marking a test: Use the `@pytest.mark.<name>` decorator.

```python
import pytest
import time

@pytest.mark.slow
def test_very_long_simulation():
    time.sleep(5)
    # ... assertions ...

@pytest.mark.integration
def test_full_pipeline():
    # ... runs multiple components ...
```

Running marked tests: Use the `-m` command-line flag.

```bash
# Run only the slow tests
pytest -m slow

# Run all tests EXCEPT the slow ones
pytest -m "not slow"

# Run integration tests
pytest -m integration
```

> Markers are essential for managing a mixed test suite of fast unit tests and slow integration or performance tests. You must register custom markers in your `pytest.ini` or `pyproject.toml` to avoid warnings.

Testing Deterministic Code

Let's test the `Vector` class from **Lecture 4**.

```python
# tests/test_vector.py
from my_package import Vector

def test_vector_addition():
    v1 = Vector(2, 3)
    v2 = Vector(10, 20)
    # Test the __add__ dunder method
    assert v1 + v2 == Vector(12, 23)

def test_vector_magnitude():
    v = Vector(3, 4)
    # Test the @property
    assert v.magnitude == 5.0
```

> Unit tests verify small, isolated pieces of your code. Testing dunder methods and properties is a great way to ensure your custom classes behave as expected.

Testing for Expected Failures

A robust test suite verifies not only that your code works, but also that it fails correctly and predictably.

`pytest.raises` is a context manager that asserts a block of code raises a specific exception.

```python
import pytest
from my_package import Vector

def test_vector_invalid_type_raises_error():
    # This test PASSES if a TypeError is raised inside the `with` block.
    # It FAILS if no exception is raised, or if a different exception is raised.
    with pytest.raises(TypeError):
        Vector(2, "three")
```

> Testing for expected exceptions is crucial for validating input handling and ensuring your code's public API is robust.

Testing Numerical Code: The Problem with Floats

Floating-point math is not exact. `assert 0.1 + 0.2 == 0.3` will **fail**.

The Problem:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Because numbers like `0.1` cannot be represented exactly in binary, small precision errors accumulate.

> Never use `==` for direct comparison of floating-point numbers in tests.

Testing Numerical Code: The Solution

The solution is to test whether numbers are "close enough" within a given tolerance.

For scalars: `pytest.approx()`

```
def test_my_calculation():
  result = 0.1 + 0.2
  assert result == pytest.approx(0.3)
```

For NumPy arrays: `numpy.testing.assert_allclose()`

```
import numpy as np

def test_my_array_calculation():
  result_array = np.array([0.1, 0.1]) + np.
array([0.2, 0.2])
  expected_array = np.array([0.3, 0.3])
  np.testing.assert_allclose(result_array,
expected_array)
```

> `np.testing.assert_allclose` provides a much more informative error message on failure than a simple `assert np.allclose(...)`, showing you exactly where the arrays differ.

Testing Stochastic Algorithms

How do you test a Monte Carlo simulation that gives a different answer every time?

By controlling the randomness. This reinforces the pattern from **Lecture 3**.

```python
# my_simulation.py
import numpy as np

def run_monte_carlo(n_samples, rng):
    # This function is now deterministic if the rng is seeded
    samples = rng.random(size=n_samples)
    return samples.mean()

# tests/test_simulation.py
def test_monte_carlo_reproducibility():
    # Create a seeded generator for the test
    rng = np.random.default_rng(42)
    result1 = run_monte_carlo(1000, rng)

    # Re-create the same generator
    rng = np.random.default_rng(42)
    result2 = run_monte_carlo(1000, rng)

    assert result1 == pytest.approx(result2)
```

> By passing a seeded RNG object, you make your stochastic function deterministic and thus testable.

Regression Testing with "Golden Files"

For very complex models, it can be hard to know the "right" answer. A **regression test** ensures the answer doesn't change unexpectedly.

The Workflow:

1. Run your model once and save the output to a "golden file" (e.g., `results.csv`, `output.npy`).
2. Commit this golden file to your repository.
3. Write a test that runs the model and asserts that the new output is identical (or numerically close) to the golden file.

```python
# tests/test_full_model.py
def test_model_regression():
    # Load the trusted result
    golden_data = np.load(
        "tests/golden_files/model_output.npy"
    )

    # Run the model now
    current_data = run_full_model()

    # Assert output hasn't changed
    np.testing.assert_allclose(
        current_data, golden_data
    )
```

> This prevents you from silently breaking your model's output during a refactor. Hashing the output file is another valid strategy for ensuring bit-for-bit identical output, especially for non-numeric files like images.

Performance Regression Testing with pytest-benchmark

How do you ensure a code change doesn't negatively impact performance? By writing a benchmark test.

The pytest-benchmark plugin (pip install pytest-benchmark) lets you test the execution time of your functions by passing a special benchmark argument to your test.

```python
def test_simulation_performance(benchmark):
    # The `benchmark` argument is a special function provided by the plugin
    # that runs your code many times to get a reliable timing.
    result = benchmark(run_full_simulation, n_steps=1000)

    # You can add assertions on the timing if needed
    assert result is not None
```

The Workflow:

1. pytest --benchmark-save=my_results saves the current performance baseline.
2. After making changes, run pytest --benchmark-compare=my_results.
3. pytest-benchmark will report any statistically significant performance regressions.

> This turns performance from a subjective concern into a testable requirement.

Fixtures

A **fixture** is a function that provides a fixed baseline state or data for your tests.

Why are fixtures powerful?

- **Reusability:** Define setup logic once and reuse it across many tests.
- **Efficiency:** Fixtures can be cached to run only once per session.
- **Readability:** A test's dependencies are declared in its signature.
- **Separation of Concerns:** Keeps setup logic separate from test logic.

```python
import pytest

@pytest.fixture
def large_dataset():
    """A fixture to load a dataset."""
    print("\n(Setting up large_dataset...)")
    data = np.load("my_data.npy")
    return data

def test_mean(large_dataset):
    # The `large_dataset` argument tells
    # pytest to run the fixture
    assert large_dataset.mean() > 0

def test_std_dev(large_dataset):
    assert large_dataset.std() > 1.0
```

> `pytest` runs the fixture and passes its return value to the test. By default, the fixture is re-run for each test function, ensuring isolation. You can change the `scope` to make it run only once per session for efficiency.

Sharing Fixtures: conftest.py

How do you share fixtures across multiple test files without importing them?

The `conftest.py` file:

- `pytest` treats files named `conftest.py` as a special local plugin.
- Fixtures defined in a `conftest.py` are automatically discovered by all tests in that directory and its subdirectories.
- You do **not** need to import fixtures from `conftest.py`.

`tests/conftest.py`

```python
import pytest
import numpy as np

@pytest.fixture(scope="session")
def shared_dataset():
    """
    Load data once for the whole
    test session.
    """
    return np.load("shared_data.npy")
```

`tests/test_analysis.py`

```python
# No import needed!
def test_analysis_mean(shared_dataset):
    assert shared_dataset.mean() > 0

# tests/subfolder/test_stats.py
def test_stats_std(shared_dataset):
    assert shared_dataset.std() > 1.0
```

> `conftest.py` is the standard, idiomatic way to provide shared fixtures and other hooks to your entire test suite.

Parameterization

What if you want to run the same test with multiple different inputs? Use `@pytest.mark.parametrize`.

```
import pytest

@pytest.mark.parametrize("test_input, expected", [
    (2, 4),
    (0, 0),
    (-3, 9),
    (3.0, 9.0),
])
def test_square(test_input, expected):
    assert square(test_input) == expected
```

`pytest` will run this test four separate times, once for each pair of inputs. If one fails, it will report exactly which case was problematic.

> This is a clean and powerful way to test a function against a range of representative inputs and edge cases.

Property-Based Testing with Hypothesis

Instead of testing for specific results, **property-based testing** checks that your code obeys certain rules or *invariants* for a wide range of inputs.

The hypothesis library generates hundreds of examples to find edge cases that break your assumptions.

```python
from hypothesis import given
from hypothesis.strategies import text

@given(text())
def test_decode_inverts_encode(s):
    # Property: decoding an encoded string should give back the original
    assert decode(encode(s)) == s
```

> hypothesis will throw everything at this test—empty strings, weird Unicode, etc.—to try to find a counterexample. This is a powerful way to increase confidence in your code's correctness.

How Hypothesis Works

`hypothesis` provides a powerful new way to think about testing.

- `@given(...)`: A decorator that runs your test function many times with different data.
- `strategies`: Objects that describe how to generate data. `hypothesis` provides strategies for integers, floats, text, NumPy arrays, and much more.
- **Shrinking**: When `hypothesis` finds a failing example, it doesn't just report it. It tries to produce the *simplest possible* failing example, which makes debugging much easier.

> Property-based testing is excellent for finding edge cases in mathematical functions, data processing pipelines, and anywhere you can define a general rule that should always hold true.

Mocking External Systems

What if your code depends on something slow, unreliable, or complex, like a network API or a large database?

Mocking lets you replace these external dependencies with a "fake" object that you control during a test.

Why use mocks?

- **Speed:** Replace a 5-minute calculation with an instant result.
- **Reliability:** Test network code without an internet connection.
- **Isolation:** Ensure your test only fails if *your* code is wrong, not because an external API is down.
- **Testability:** Easily simulate specific scenarios (like API errors or empty returns) that are hard to trigger with the real system.

Example with `pytest-mock`:

```python
# tests/test_analysis.py
def test_process_data(mocker):
    # Replace the real API call
    fake_data = {"value": [1, 2, 3]}
    mocker.patch(
        "my_analysis.get_data_from_api",
        return_value=fake_data
    )

    # The function under test now
    # uses the fake data
    result = my_analysis.process_data()
    assert result == 2.0
```

> The `mocker.patch` function temporarily replaces the `get_data_from_api` function inside the `my_analysis` module with a "fake" object that returns a pre-defined value. The original function is restored after the test completes.

Inspecting Mock Objects

The object returned by `mocker.patch` is a special `Mock` object that records how it was used. You can make assertions about these interactions.

This allows you to test *behavior* (e.g., "was the API called with the right ID?") not just the final result.

```python
def test_process_data_calls_api_correctly(mocker):
    # Patch the API call and get a reference to the mock object
    mock_api_call = mocker.patch(
        "my_analysis.get_data_from_api",
        return_value={"value": [1, 2, 3]}
    )

    # Run the function that is supposed to call the API
    my_analysis.process_data(user_id="test_user")

    # Assert that the mock was called correctly
    mock_api_call.assert_called_once()
    mock_api_call.assert_called_once_with(user_id="test_user")
```

> This pattern is powerful for testing functions with side effects, where the important outcome isn't the return value but the interactions it has with other systems.

Measuring Effectiveness: Code Coverage

Code coverage, provided by the pytest-cov plugin (pip install pytest-cov), measures which lines of your source code were executed by your test suite.

To run:

```
pytest --cov=my_package
```

Example Output:

```
---------- coverage: platform linux, python 3.10.12-final-0 -----------
Name                    Stmts   Miss  Cover
-----------------------------------------
src/my_package/a.py        20      2    90%
src/my_package/b.py        12     12     0%
-----------------------------------------
TOTAL                      32     14    56%
```

> Coverage is a useful tool to find untested parts of your code, but it's not a measure of test *quality*. 100% coverage doesn't mean your code is bug-free.

Part 3: From Testing to Automation (CI/CD)

The Motivational Bridge

Our tests are more than just a safety net; they are a core part of the development and debugging workflow.

- **Test-Driven Debugging:** When a bug is found, write a failing test that reproduces it *first*. Then, fix the code until the test passes.
- **Collaboration:** A CI pipeline ensures that all changes are validated against the full test suite in a clean, standard environment.

> Continuous Integration (CI) automates this process, making it the cornerstone of reliable team-based software development.

What is CI/CD?

CI: Continuous Integration

- The practice of frequently merging all developer's code changes into a central repository.
- After each merge, an **automated build and test** process is triggered.
- **Goal:** Find and fix integration bugs early.

CD: Continuous Delivery / Deployment

- The practice of automatically deploying every change that passes the CI stage to a testing or production environment.
- **Goal:** Release new features to users quickly and reliably.

> Automating the publication of a package to PyPI is a form of **Continuous Delivery**. We will build a workflow for both CI (testing) and CD (publishing).

Introduction to GitHub Actions

GitHub Actions is a CI/CD platform built directly into GitHub. It lets you automate workflows in response to `git` events.

- **Workflow:** An automated process defined in a YAML file.
- **Event:** A trigger for the workflow (e.g., `push`, `pull_request`).
- **Job:** A set of steps that run on a virtual machine.
- **Runner:** The virtual machine that executes a job (e.g., `ubuntu-latest`, `windows-latest`).

> This directly connects our **Lecture 2** `git` workflow to an automated process. A `git push` can now automatically trigger a test run.

Part 4: GitHub Actions in Practice

Anatomy of a Workflow File

Workflows are defined in YAML files inside the `.github/workflows/` directory of your repository.

A workflow is composed of one or more **jobs**, which run in parallel by default. Each job runs on a fresh **runner** and is made up of a sequence of **steps**. Steps can either run shell commands (`run`) or use pre-built community actions (`uses`).

```yaml
# .github/workflows/ci.yml
name: Run Tests

# 1. Trigger on events
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

# 2. Define jobs
jobs:
  build:
    # 3. Specify runner
    runs-on: ubuntu-latest
    # 4. List steps to execute
    steps:
      - name: Check out code
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.10'

      - name: Run a command
        run: echo "Hello, world!"
```

The GitHub Actions Ecosystem

- **Runners:** These are the virtual machines that run your jobs.
  - **GitHub-hosted:** Convenient, managed by GitHub (`ubuntu-latest`, `windows-latest`, etc.).
  - **Self-hosted:** Your own machines that you register with GitHub. Necessary for tasks requiring special hardware (like GPUs) or for security compliance.
- **Actions Marketplace:** The `uses: actions/checkout@v4` syntax calls a reusable action from the marketplace. This saves you from writing boilerplate code. `actions/checkout` is the standard action for checking out your repository's code onto the runner.
- **Custom Actions:** You can write your own reusable actions, but for most use cases, running shell commands with `run:` is sufficient and more explicit.

Using Variables and Secrets

Workflows often need access to variables or sensitive information.

- **Environment Variables:** Use the `env` key to set variables for a job or step.

```yaml
steps:
  - name: Run custom script
    env:
      MY_VARIABLE: "some_value"
    run: ./my_script.sh
```

- **Secrets:** For sensitive data like API keys, use GitHub Secrets.

  1. Add the secret in your repository settings (`Settings` > `Secrets and variables` > `Actions`).
  2. Access it in your workflow using the `secrets` context: `${{ secrets.MY_SECRET_NAME }}`.

> Secrets are encrypted and are never printed in logs, making them the secure way to handle credentials.

Example 1: Automated Testing

Let's create a workflow to install dependencies and run `pytest`.

```yaml
# ... (on, jobs, runs-on)
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -e .[test]

      - name: Run pytest
        run: pytest
```

> Now, every push or PR to `main` will automatically trigger a full test run in a clean environment, preventing broken code from being merged.

Enforcing Quality with Branch Protection

Now that we have an automated test suite, we can configure GitHub to *require* it to pass before any code is merged into our `main` branch. This is done with **branch protection rules**.

Common Rules for the `main` branch:

- **Require status checks to pass before merging:** This is the key. It blocks PRs if your `pytest` workflow fails.
- **Require pull request reviews:** Prevent merging without at least one other person's approval.
- **Do not allow force pushes:** Protects the integrity of the branch history.

> Branch protection turns your CI pipeline into a quality gate, ensuring the `main` branch is always stable and tested.

Validating Portability with Matrix Builds

Scientific code must be portable. A **matrix build** runs your tests across multiple configurations.

```yaml
jobs:
  test:
    runs-on: ${{ matrix.os }} # Use the OS from the matrix
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        python-version: ['3.9', '3.10', '3.11']

    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }} # Use Python from matrix
      # ... install and test steps
```

> This job will now spawn 3x3 = 9 parallel runs, ensuring your code works across different operating systems and Python versions.

Debugging with Artifacts

When a test that *validates* a generated output (like a plot or data file) fails in CI, you need to see the incorrect output to debug it.

You can save files from the CI run as **artifacts**.

```yaml
      - name: Run analysis and generate plot
        run: python run_analysis.py

      # This step runs if any previous step in the job fails
      - name: Upload plot artifact on failure
        if: failure()
        uses: actions/upload-artifact@v4
        with:
          name: failed-plot
          path: analysis_plot.png
```

> If the job fails, the generated `analysis_plot.png` is uploaded. You can download it from the GitHub Actions summary page to inspect what went wrong.

Show Your Status: Badges

Badges are a standard way to communicate project health at a glance in your `README.md`. Other common badges include code coverage and package version.

The Markdown:

```
![CI Status](https://github.com/<USER>/<REPO>/actions/workflows/ci.yml/badge.svg)
```

> This URL is a special endpoint provided by GitHub. It generates a live SVG image reflecting the status of the workflow defined in `ci.yml`.

Example 2: Automated Publishing to PyPI

This workflow is the capstone of our automation journey, building on **Lecture 5**.

The Goal: Automatically publish your package to PyPI whenever you create a new version tag in `git`.

```yaml
name: Publish to PyPI

on:
  push:
    tags:
      - 'v*.*.*' # Trigger on tags like v1.0.0

jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      # ... setup python ...
      - name: Build package
        run: python -m build
      - name: Publish to PyPI
        uses: pypa/gh-action-pypi-publish@release/v1
        with:
          password: ${{ secrets.PYPI_API_TOKEN }}
```

> The `secrets.PYPI_API_TOKEN` is a secure variable you configure in your GitHub repository settings. This workflow fully automates your release process.

Example 3: Scheduled Jobs with cron

Workflows don't have to be triggered by git events. You can run them on a schedule using standard cron syntax.

This is perfect for tasks like:

- Daily web scraping to collect new data.
- Weekly model retraining.
- Generating a nightly report.

```yaml
name: Nightly Data Scrape

on:
  schedule:
    # Runs at 05:30 UTC every day from Monday to Friday
    - cron: '30 5 * * 1-5'

jobs:
  scrape:
    runs-on: ubuntu-latest
    steps:
      # ... checkout, setup python, run script, commit new data ...
```

> The `cron` string is a standard format: `minute hour day(month) month day(week)`. GitHub Actions provides an easy way to automate recurring computational tasks.

Resources and Further Reading

pytest

- **Official Documentation** (*https://docs.pytest.org/en/stable/*)
- **Fixtures Guide** (*https://docs.pytest.org/en/stable/explanation/fixtures.html*): A deep dive into pytest 's most powerful feature.
- pytest-benchmark **Docs** (*https://pytest-benchmark.readthedocs.io/en/latest/*): For performance testing.

hypothesis

- **Official Documentation** (*https://hypothesis.readthedocs.io/en/latest/*): Find bugs you never thought to look for.

GitHub Actions

- **Official Documentation** (*https://docs.github.com/en/actions*): The definitive source.
- **Workflow Syntax Guide** (*https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions*): All the YAML options explained.
- **Building and Testing Python** (*https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-python*): Official examples.

Final Demo: Tying It All Together

We will now apply these concepts to our `card_games` package in a live demo.

You can follow along or review the code at: **https://code.harvard.edu/AM215/blackjack-testing-demo**

The Plan:

1. **Write & Run Tests:** Add unit tests for `get_hand_value` and establish a performance baseline with `pytest-benchmark`.
2. **Verify Refactoring:** Show how a performance refactor can be verified with a benchmark test.
3. **Automate with CI:** Create a GitHub Actions workflow to run tests automatically on every push.
4. **Enforce Quality:** Use branch protection rules to block a pull request with failing tests.
5. **Automate Publishing:** Create a workflow to automatically publish the package to TestPyPI when a version tag is pushed.