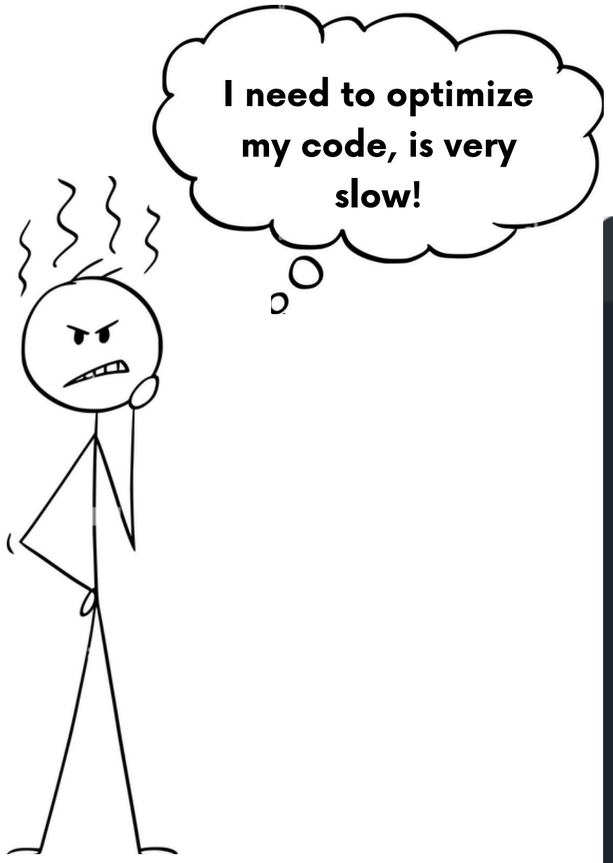
TEST, TEST AND TEST AGAIN
INTRO TO PYTEST

Development Tools for Scientific Computing course.
Academic Year 2024-2025.

Pasquale Claudio Africa, Dario Coscia

SISSA, TRIESTE

WHY TESTING?



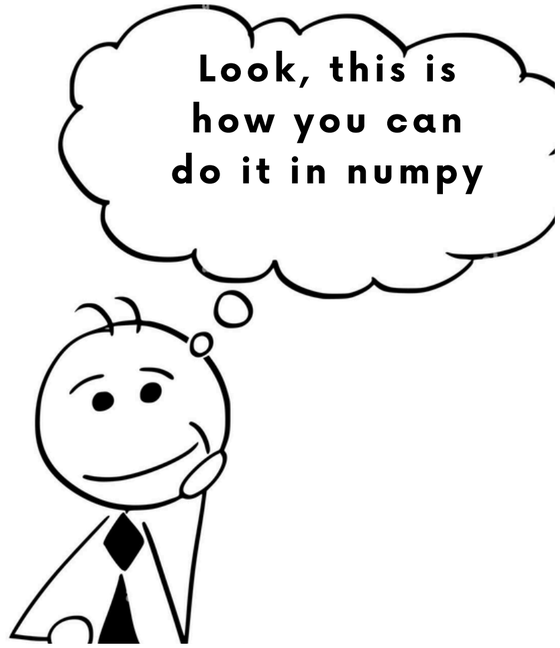
cool_function.py

```python
def matrix_mult(A: np.ndarray, B: np.ndarray) → np.ndarray:
    # Check if the matrices can be multiplied
    if A.shape[1] ≠ B.shape[0]:
        raise ValueError

    result = np.zeros((A.shape[0], B.shape[1]))

    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(A.shape[1]):
                result[i, j] += A[i, k] * B[k, j]

    return result
```

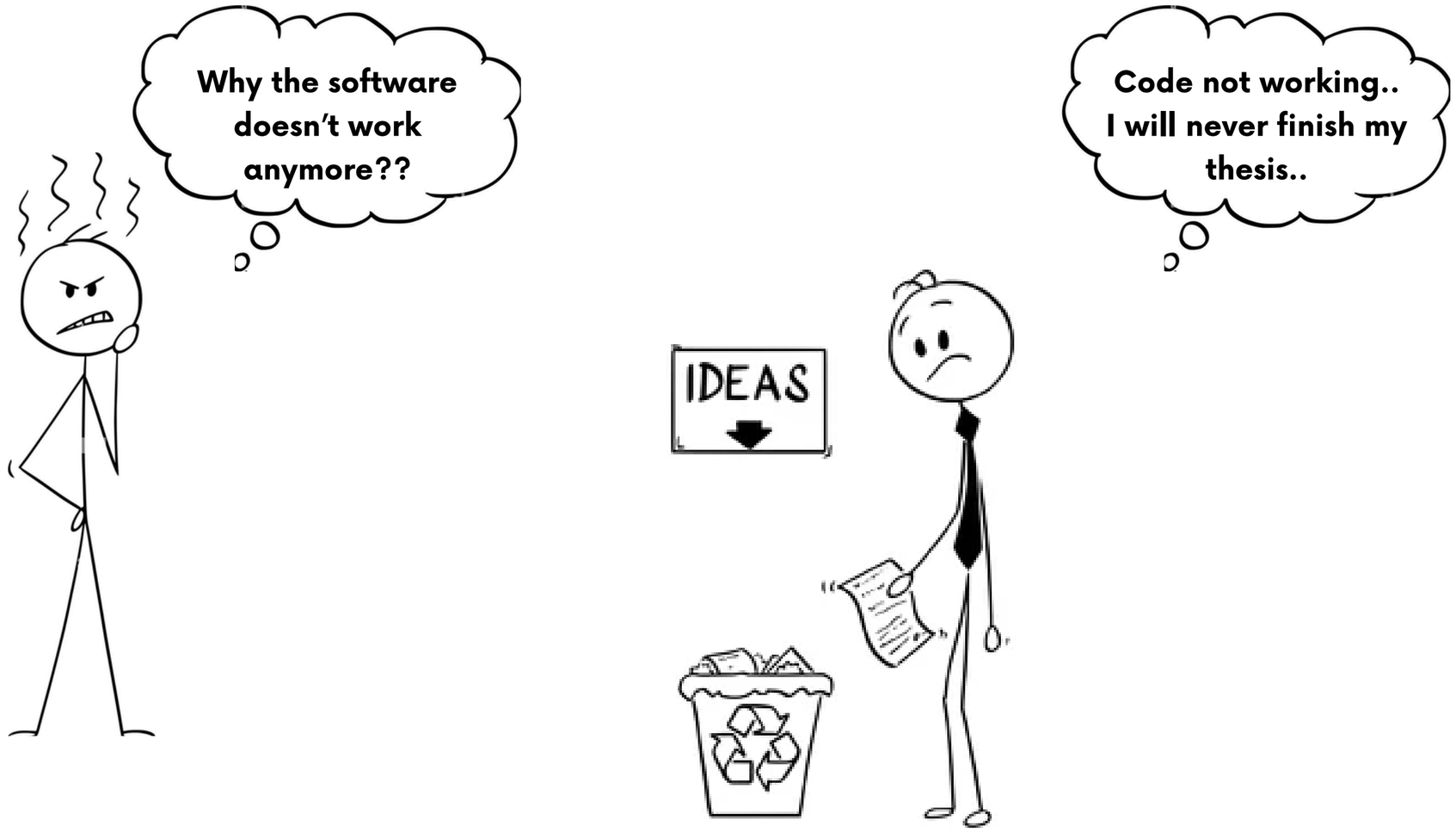
WHY TESTING?

cool_function.py

```python
def matrix_mult(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    # Check if the matrices can be multiplied
    if A.shape[1] != B.shape[0]:
        raise ValueError
    result = np.dot(A, B)
```



MEANWHILE USERS....



WHAT HAPPENED?

Buggy code

cool_function.py

```python
def matrix_mult(A: np.ndarray, B: np.ndarray) → np.ndarray:
    # Check if the matrices can be multiplied
    if A.shape[1] ≠ B.shape[0]:
        raise ValueError
    result = np.dot(A, B)
```

missing return statement

Correct code

cool_function.py

```python
def matrix_mult(A: np.ndarray, B: np.ndarray) → np.ndarray:
    # Check if the matrices can be multiplied
    if A.shape[1] ≠ B.shape[0]:
        raise ValueError
    result = np.dot(A, B)
    return results
```

PYTEST

pytest is a popular testing framework for Python that is **used to write and run tests** for Python code. It is highly flexible, easy to use, and supports a variety of testing functionalities.

```python
# content of test_sample.py
def func(x):
    return x + 1


def test_answer():
    assert func(3) == 5
```

The test

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-8.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_sample.py F                                                  [100%]

================================= FAILURES =================================
_____________________________ test_answer _____________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
======================== short test summary info ========================
FAILED test_sample.py::test_answer - assert 4 == 5
=========================== 1 failed in 0.12s ===========================
```

RUN PYTEST

In general, **pytest** is invoked with the command pytest. This will execute all tests in all files whose names follow the form **test_*.py** or ***_test.py** in the current directory and its subdirectories.

Run tests in a module

```
pytest test_mod.py
```

Run tests in a directory

```
pytest testing/
```

To run a specific test within a module:

```
pytest tests/test_mod.py::test_func
```

To run all tests in a class:

```
pytest tests/test_mod.py::TestClass
```

Specifying a specific test method:

```
pytest tests/test_mod.py::TestClass::test_method
```

MARK TESTS

By using the **pytest.mark** helper you can easily set metadata on your test functions. You can list all the markers, including builtin and custom, using the CLI - **pytest --markers**.

Here are some of the builtin markers:

- usefixtures - use fixtures on a test function or class
- skip - always skip a test function
- skipif - skip a test function if a certain condition is met
- xfail - produce an “expected failure” outcome if a certain condition is met
- parametrize - perform multiple calls to the same test function.
- ....

PARAMETRIZED TESTS


```python
# Parametrized test cases for successful matrix multiplication
@pytest.mark.parametrize(
    "A, B, expected",
    [
        # Test case 1: Simple 2x2 matrix multiplication
        (np.array([[1, 2], [3, 4]]), np.array([[5, 6], [7, 8]]), np.array([[19, 22], [43, 50]])),

        # Test case 2: 2x3 and 3x2 matrix multiplication
        (np.array([[1, 4, 6], [2, 3, 1]]), np.array([[7, 8], [9, 10], [11, 12]]), np.array([[109, 120], [50, 56]])),

        # Test case 3: Identity matrix multiplication
        (np.array([[1, 0], [0, 1]]), np.array([[5, 6], [7, 8]]), np.array([[5, 6], [7, 8]])),

        # Test case 4: Zero matrix multiplication
        (np.array([[0, 0], [0, 0]]), np.array([[1, 2], [3, 4]]), np.array([[0, 0], [0, 0]])),

        # Test case 5: 3x3 matrix multiplication
        (np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
         np.array([[9, 8, 7], [6, 5, 4], [3, 2, 1]]),
         np.array([[30, 24, 18], [84, 69, 54], [138, 114, 90]])),
    ]
)
def test_matrix_mult(A, B, expected):
    result = matrix_mult(A, B)
    # Check if the result is close to the expected value
    assert np.array_equal(result, expected), f"Expected {expected}, but got {result}"
```

ERROR HANDLING TESTS


```python
# Parametrized test cases for ValueError when matrix dimensions are incompatible
@pytest.mark.parametrize(
    "A, B",
    [
        # Test case 1: 2x3 and 2x2 matrices (incompatible)
        (np.array([[1, 2, 3], [4, 5, 6]]), np.array([[1, 2], [3, 4]])),

        # Test case 3: 1x3 and 4x2 matrices (incompatible)
        (np.array([[1, 2, 3]]), np.array([[1, 2], [3, 4], [5, 6], [7, 8]])),
    ]
)
def test_matrix_mult_value_error(A, B):
    # Check if ValueError is raised for incompatible matrices
    with pytest.raises(ValueError):
        matrix_mult(A, B)
```

WHEN AND WHAT SHOULD I TEST

- **Before You Start Coding (Test-Driven Development - TDD):**
  - Why: In TDD, you write tests before writing the actual code. This approach helps define clear requirements and ensures the code is always covered by tests.
  - When: As you start implementing a new feature, function, or class. You first write a failing test, then implement the functionality, and finally, refactor to ensure everything passes.
- **During Development:**
  - Why: It's important to write tests while developing your code to catch errors early, prevent regressions, and ensure your logic works as expected.
  - When: Write tests after you implement a feature or piece of logic. It’s recommended to test small units of code, such as functions or methods, right after writing them.

WHEN AND WHAT SHOULD I TEST

- **Before Merging Code:**
  - Why: Before merging a feature branch or submitting a pull request (PR), running tests ensures that the new code doesn't break anything already working in the project.
  - When: Once you've completed a feature and are ready to integrate it, run your tests to verify that everything works and no functionality is broken.
- **After Refactoring**:
  - Why: When refactoring code (e.g., simplifying, optimizing, or restructuring it), you want to make sure that existing functionality is not broken.
  - When: After refactoring, rerun your tests to check that the behavior hasn't changed inadvertently.

MORE ON TESTING

- **Official pytest Documentation:**
  - pytest Documentation (https://docs.pytest.org/en/6.2.x/contents.html)
  - This is the official documentation for pytest, which is the most popular framework for unit testing in Python. It provides comprehensive information on how to write unit tests, fixtures, and test suites using pytest.
  -
- **Test-Driven Development with Python:**
  - book: Test-Driven Development with Python (https://www.obeythetestinggoat.com/)
  - This is a fantastic resource if you're learning Test-Driven Development (TDD). It's written by Harry J.W. Percival and is great for Python developers who want to master unit testing and apply TDD practices.
  -