Lab 5: Write Unit Tests and Automate Testing with GitHub Actions

Lab overview

In this lab, you will explore writing comprehensive unit tests using PyTest and automating the testing process with GitHub Actions. You'll learn to create modular test functions, understand PyTest test structure and conventions, set up continuous integration (CI) pipelines, and interpret test results and failure logs. By the end of this lab, you'll have a fully tested Python application with automated testing that runs on every code change.

In this lab, you will:

- Write clear and modular unit tests using PyTest
- Automate testing via CI on push/pull_request events
- Interpret GitHub Actions logs and test failures
- Handle test coverage and reporting
- Fix common import path issues in testing

Estimated completion time

60 minutes

Task 1: Creating a Python application to test

In this task, you will create a simple Python utility application with multiple functions that can be thoroughly tested.

1. To create a folder named **Lab5** on the Desktop, right-click on the Desktop, click **New**, then select **Folder**. Name the folder **Lab5**. Open with **Visual Studio Code.**
2. To create the main application file, in VSCode, click on the **Explorer** icon in the left sidebar (or press **Ctrl+Shift+E**).
3. Click the **New File** icon next to the Lab5 folder name
4. Name the file **math_utils.py**.
5. Click on **math_utils.py** to open it in the editor.
6. Add the following code.

```
"""Simple Math Utilities for Testing Demo"""


def add(a, b):
    """Add two numbers."""
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Arguments must be numbers")
    return a + b


def multiply(a, b):
    """Multiply two numbers."""
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Arguments must be numbers")
    return a * b


def divide(a, b):
    """Divide first number by second."""
```

```
    if not isinstance(a, (int, float)) or not isinstance(b, (int,
float)):
        raise TypeError("Arguments must be numbers")
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b


def is_even(number):
    """Check if number is even."""
    if not isinstance(number, int):
        raise TypeError("Argument must be an integer")
    return number % 2 == 0
```

7. Save the file by pressing Ctrl+S.
8. To create a string utility module, click the New File icon again.
9. Name the file string_utils.py.
10. Add the following code.

```
"""String Utilities for Testing Demo"""


def reverse_string(text):
    """Reverse a string."""
    if not isinstance(text, str):
        raise TypeError("Argument must be a string")
    return text[::-1]


def count_words(text):
    """Count words in a string."""
    if not isinstance(text, str):
```

```
        raise TypeError("Argument must be a string")
    return len(text.split()) if text.strip() else 0


def is_palindrome(text):
    """Check if string is palindrome (ignoring case)."""
    if not isinstance(text, str):
        raise TypeError("Argument must be a string")
    cleaned = text.lower().replace(" ", "")
    return cleaned == cleaned[::-1]
```

11. Save the file by pressing Ctrl+S.
12. To create requirements file, click the New File icon.
13. Name the file requirements.txt.
14. Add the following content.

```
pytest==7.4.3
pytest-cov==4.1.0
```

15. Save the file by pressing Ctrl+S.
16. Verify your file structure. Your VSCode Explorer should now show:

```
Lab5/
├── math_utils.py
├── string_utils.py
└── requirements.txt
```

Task 2: Writing unit tests with PyTest

In this task, you will create comprehensive unit tests for your Python functions using PyTest framework.

1. To install PyTest and dependencies, open VSCode terminal (Terminal > New Terminal).
2. Ensure you're in the Lab5 directory.
3. Install the testing dependencies.

```
# Install pip if not available

sudo apt update

sudo apt install python3-pip -y


# Install pytest and coverage tools

pip3 install pytest pytest-cov


# Verify installation

pytest --version
```

4. To create test directory structure, in VSCode Explorer, click New Folder.
5. Name it tests.
6. Right-click on tests folder.
7. Select New File.
8. Name it init.py. This makes it a Python package.
9. Leave the file empty and save it.
10. To create math utilities tests, right-click on tests folder.
11. Select New File.
12. Name it test_math_utils.py.
13. Add the following test code.

```
"""Tests for math_utils module"""


import pytest

from math_utils import add, multiply, divide, is_even
```

```
class TestBasicOperations:
    """Test basic math operations."""

    def test_add_positive(self):
        """Test addition with positive numbers."""
        assert add(2, 3) == 5
        assert add(10, 15) == 25

    def test_add_negative(self):
        """Test addition with negative numbers."""
        assert add(-2, -3) == -5
        assert add(-10, 5) == -5

    def test_multiply_basic(self):
        """Test multiplication."""
        assert multiply(3, 4) == 12
        assert multiply(-2, 5) == -10
        assert multiply(0, 100) == 0

    def test_divide_basic(self):
        """Test division."""
        assert divide(10, 2) == 5
        assert divide(15, 3) == 5
        assert divide(-10, 2) == -5
```

```
class TestErrorHandling:
    """Test error conditions."""

    def test_add_type_error(self):
        """Test add with invalid types."""
        with pytest.raises(TypeError):
            add("2", 3)
        with pytest.raises(TypeError):
            add(2, None)

    def test_divide_by_zero(self):
        """Test division by zero."""
        with pytest.raises(ValueError, match="Cannot divide by zero"):
            divide(10, 0)

    def test_is_even_type_error(self):
        """Test is_even with invalid type."""
        with pytest.raises(TypeError):
            is_even(3.5)


class TestEvenNumbers:
    """Test even number checking."""
```

```
    def test_is_even_true(self):
        """Test even numbers."""
        assert is_even(2) is True
        assert is_even(0) is True
        assert is_even(100) is True

    def test_is_even_false(self):
        """Test odd numbers."""
        assert is_even(1) is False
        assert is_even(3) is False
        assert is_even(99) is False


# Parametrized tests
@pytest.mark.parametrize("a,b,expected", [
    (1, 2, 3),
    (0, 0, 0),
    (-1, 1, 0),
    (10, -5, 5)
])
def test_add_multiple_inputs(a, b, expected):
    """Test add with multiple parameter sets."""
    assert add(a, b) == expected


@pytest.mark.parametrize("number,expected", [
```

```
    (0, True),
    (2, True),
    (1, False),
    (3, False)
])
def test_is_even_multiple_inputs(number, expected):
    """Test is_even with multiple inputs."""
    assert is_even(number) == expected
```

14. Save the file by pressing Ctrl+S.
15. To create string utility tests, right-click on tests folder.
16. Select New File.
17. Name it test_string_utils.py.
18. Add the following test code.

```
"""Tests for string_utils module"""

import pytest
from string_utils import reverse_string, count_words, is_palindrome


class TestStringOperations:
    """Test string manipulation functions."""

    def test_reverse_basic(self):
        """Test string reversal."""
        assert reverse_string("hello") == "olleh"
        assert reverse_string("") == ""
```

```
        assert reverse_string("a") == "a"


    def test_count_words_basic(self):
        """Test word counting."""
        assert count_words("hello world") == 2
        assert count_words("Python is awesome") == 3
        assert count_words("") == 0
        assert count_words("   ") == 0


    def test_palindrome_basic(self):
        """Test palindrome detection."""
        assert is_palindrome("racecar") is True
        assert is_palindrome("hello") is False
        assert is_palindrome("") is True
        assert is_palindrome("A") is True


    def test_palindrome_case_insensitive(self):
        """Test palindrome with mixed case."""
        assert is_palindrome("Racecar") is True
        assert is_palindrome("A man a plan a canal Panama") is True


class TestStringErrors:
    """Test string function error handling."""


    def test_reverse_type_error(self):
```

```
        """Test reverse with invalid type."""
        with pytest.raises(TypeError):
            reverse_string(123)
        with pytest.raises(TypeError):
            reverse_string(None)

    def test_count_words_type_error(self):
        """Test count_words with invalid type."""
        with pytest.raises(TypeError):
            count_words(123)

    def test_palindrome_type_error(self):
        """Test is_palindrome with invalid type."""
        with pytest.raises(TypeError):
            is_palindrome(123)


# Parametrized tests
@pytest.mark.parametrize("text,expected", [
    ("hello", "olleh"),
    ("Python", "nohtyP"),
    ("12345", "54321"),
    ("", "")
])
def test_reverse_multiple_inputs(text, expected):
    """Test reverse with multiple inputs."""
```

```
    assert reverse_string(text) == expected


@pytest.mark.parametrize("text,count", [
    ("hello world", 2),
    ("Python is great", 3),
    ("single", 1),
    ("", 0)
])
def test_count_words_multiple_inputs(text, count):
    """Test count_words with multiple inputs."""
    assert count_words(text) == count
```

19. Save the file by pressing Ctrl+S.
20. Update your file structure. Your directory should now look like the following.

```
Lab5/
├── tests/
│   ├── __init__.py
│   ├── test_math_utils.py
│   └── test_string_utils.py
├── math_utils.py
├── string_utils.py
└── requirements.txt
```

Task 3: Running tests locally

In this task, you will run your tests locally and interpret the results.

1. To run all tests with the correct Python path, in your VSCode terminal, make sure you're in the Lab5 directory.
2. Run tests with the Python path fix.

> **Note**
>
> There is a space between the dot and pytest!

```
# Set Python path and run tests
PYTHONPATH=. pytest


# Alternative method
python -m pytest
```

You should see output showing all tests passing.

3. Run tests with verbose output.

```
# Show detailed test results
python -m pytest -v
```

> or alternatively [please note the spaces]

```
PYTHONPATH=. pytest -v


# Show even more detailed output
python -m pytest -vv
```

4. Run specific test files.

```
 # Run only math tests
python -m pytest tests/test_math_utils.py works.


# Run only string tests
```

```
Python -m pytest tests/test_string_utils.py


# Run specific test class

PYTHONPATH=. pytest tests/test_math_utils.py::TestBasicOperations


# Run specific test function

PYTHONPATH=. pytest
tests/test_math_utils.py::TestBasicOperations::test_add_positive
```

5. Run tests with coverage reporting.

```
# Run tests and show coverage report

PYTHONPATH=. pytest --cov=math_utils --cov=string_utils


# Generate detailed coverage report

PYTHONPATH=. pytest --cov=math_utils --cov=string_utils --cov-
report=term-missing


# Generate HTML coverage report

PYTHONPATH=. pytest --cov=math_utils --cov=string_utils --cov-
report=term
```

6. Understanding PyTest output. Successful test run looks like the following.

```
==================== test session starts ====================
collected 16 items

tests/test_math_utils.py ............ [ 75%]
tests/test_string_utils.py .... [100%]

==================== 16 passed in 0.05s ====================
```

Failed test looks like the following.

```
FAILED tests/test_math_utils.py::test_add_positive
_________________ test_add_positive _________________

    def test_add_positive():
>       assert add(2, 3) == 6  # This would fail
E       assert 5 == 6
E        +  where 5 = add(2, 3)
```

7. To test different scenarios, introduce a deliberate failure to see how PyTest reports errors.
8. Open tests/test_math_utils.py.
9. Change one assertion to make it fail.

```
def test_add_positive(self):
    """Test addition with positive numbers."""
    assert add(2, 3) == 6  # Changed from 5 to 6 to cause failure
```

10. Run pytest and observe the detailed error output.

```
PYTHONPATH=. pytest -v
```

11. Fix the test back to the correct value.

12. Generate test coverage report.

```
# Create detailed coverage report

PYTHONPATH=. pytest --cov=math_utils --cov=string_utils --cov-
report=html --cov-report=html


# Open coverage report in browser (if available)

firefox htmlcov/index.html
```

13. Run tests with different output formats.

```
# Run with minimal output

PYTHONPATH=. pytest -q


# Run and stop on first failure

PYTHONPATH=. pytest -x


# Run and show local variables on failure

PYTHONPATH=. pytest -l


# Run with timing information

PYTHONPATH=. pytest --durations=5
```

Task 4: Automating tests with GitHub Actions

In this task, you will set up GitHub Actions to automatically run your tests on every push and pull request.

1. To initialize a Git repository, in VSCode terminal, run the following code.

```
git init
```

```
git add .
```

```
git commit -m "Initial commit: Python app with PyTest tests"
```

2. To create GitHub repository, open your web browser and go to **https://github.com**.
3. Log in to your GitHub account.
4. Click the **+** icon in the top right corner.
5. Select **New repository**.
6. Name the repository **python-pytest-demo**.
7. Leave it as **Public**.
8. Do NOT initialize with README.
9. Click **Create repository**.
10. To connect local repository to GitHub, copy the commands from GitHub and run in terminal (replace YOUR_GITHUB_USERNAME).

```
git branch -M main
```

```
git remote add origin https://github.com/YOUR_GITHUB_USERNAME/python-pytest-demo.git
```

```
git push -u origin main
```

11. To create GitHub Actions workflow directory, in VSCode Explorer, right-click in the **Lab5** folder.
12. Select **New Folder** and name it **.github** (include the dot at the beginning).
13. Right-click on **.github** folder and select **New Folder**.
14. Name it **workflows**.
15. To create the testing workflow file, right-click on **workflows** folder.
16. Select **New File**.
17. Name it **test.yml**.

18. Add the following workflow (includes PYTHONPATH fix).

```
name: Run Tests

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.9, '3.10', '3.11']

    steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
```

```
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run tests
      run: |
        PYTHONPATH=. pytest -v

    - name: Run tests with coverage
      run: |
        PYTHONPATH=. pytest --cov=math_utils --cov=string_utils --cov-report=term
```

19. Save the file by pressing Ctrl+S.
20. To create a README file with status badge, click the New File icon in Lab5 folder.
21. Name it README.md.
22. Add the following content (replace YOUR_GITHUB_USERNAME).

```
# Python PyTest Demo

![Tests](https://github.com/YOUR_GITHUB_USERNAME/python-pytest-demo/workflows/Run%20Tests/badge.svg)

A demonstration of Python testing with PyTest and GitHub Actions.

## Features

- Math utilities with comprehensive tests
- String utilities with error handling
- Automated testing with GitHub Actions
```

```
- Test coverage reporting
- Multi-Python version testing

## Running Tests

```bash
# Install dependencies
pip install -r requirements.txt

# Run all tests
PYTHONPATH=. pytest

# Run with coverage
PYTHONPATH=. pytest --cov=math_utils --cov=string_utils

# Run specific test file
PYTHONPATH=. pytest tests/test_math_utils.py
```

23. Replace YOUR_GITHUB_USERNAME with your actual GitHub username. Save the file.

Test structure

- tests/test_math_utils.py - Math function tests
- tests/test_string_utils.py - String function tests
- Parametrized tests for multiple inputs
- Error handling tests
- Type validation tests

Import path solution

This lab uses PYTHONPATH=. pytest to resolve module imports in tests. This ensures that test files can properly import the modules under test.
```

Coverage reports

Generate detailed coverage reports:

```
PYTHONPATH=. pytest --cov=math_utils --cov=string_utils --cov-report=html
```

24. Commit and push the workflow with the following code.

```
git add .
```

```
git commit -m "Add comprehensive testing workflow with PYTHONPATH fix"
```

```
git push origin main
```

25. To monitor the workflow execution, go to your GitHub repository in the browser.
26. Click the **Actions** tab. You should see a workflow run called Run Tests.
27. Click on the workflow to see detailed progress.
28. The workflow will run tests on Python 3.9, 3.10, and 3.11.
29. Understanding workflow results. Successful workflow shows as follows.

✅ Run Tests

├── ✅ test (3.9)

├── ✅ test (3.10)

└── ✅ test (3.11)

Each job shows:

✅ Checkout code

✅ Set up Python

✅ Install dependencies

✅ Run tests

✅ Run tests with coverage

Failed workflow shows:

- ❌ Red X marks for failed steps
- Detailed error logs showing which tests failed
- Specific line numbers and error messages

30. To test the automated workflow, make a change to introduce a test failure. Open **math_utils.py** and temporarily break a function.

```
def add(a, b):
    """Add two numbers."""
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Arguments must be numbers")
    return a + b + 1  # Add +1 to break tests
```

31. Commit and push.

```
git add math_utils.py
git commit -m "Test workflow failure detection"
git push origin main
```

32. Watch the **Actions** tab to see the workflow fail.
33. Fix the function and push again to see success.
34. Interpreting test failure logs in GitHub Actions. When tests fail, you'll see detailed output like the following.

```
FAILED tests/test_math_utils.py::TestBasicOperations::test_add_positive
_____________________ TestBasicOperations.test_add_positive _____________________

self = <tests.test_math_utils.TestBasicOperations object at 0x...>

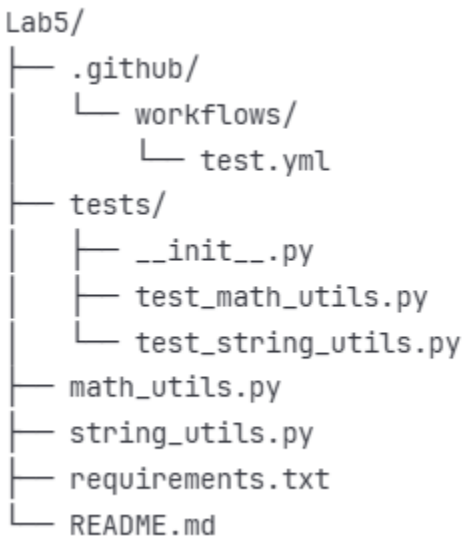
    def test_add_positive(self):
        """Test addition with positive numbers."""
```

```
>       assert add(2, 3) == 5
E       assert 6 == 5
E        +  where 6 = add(2, 3)
```

This shows:

- **Test that failed:** test_add_positive
- **Expected result:** 5
- **Actual result:** 6
- **Failing line:** assert add(2, 3) == 5

35. Common debugging steps for failed tests:
    35.1. Click on the failed job in GitHub Actions.
    35.2. Scroll to the **Run tests** step.
    35.3. Look for the specific assertion that failed.
    35.4. Check the expected vs. actual values.
    35.5. Fix the code or test as needed.
    35.6. Push the fix to retrigger the workflow.
36. Final file structure should look like the following.



Lab review

1. What is the purpose of the **pytest.raises()** context manager?
   A. To generate test data automatically
   B. To test that a function raises a specific exception
   C. To skip tests that might fail
   D. To measure test execution time

STOP

You have successfully completed this lab.