揚州大學

Yangzhou University

Experimental Report

Unit Testing & Coverage

Course name	Soft Quality Assurance & Testing
Name/Student ID	NAME: BAYZID MD (俊杰) ID: 238801246
College	College of Information Engineering
Major	Software Engineering
Class	SE 2023
Supervisor	(吴潇雪)
Date	26th October, 2025

Introduction

Project Name: AutomationPanda / python-testing-101

Project Overview

The python-testing-101 project is a Python-based calculator developed to demonstrate **software testing best practices**, including unit testing, boundary testing, exception handling, and code coverage analysis. It provides a hands-on environment to practice professional testing methodologies on a small but functional project.

The project includes both **function-based** and **class-based** implementations of a calculator, supporting the following features:

- **Basic operations:** addition, subtraction, multiplication, and division
- **Boundary and exception handling:** handling division by zero, negative number operations
- **Comparative functions:** finding the maximum and minimum values among multiple inputs
- **State management:** initializing calculator state and supporting chained operations

The primary goal of this project is to **demonstrate how to systematically test software** using Python's pytest framework and measure coverage using coverage.py. The tests ensure that all features work as expected, including edge cases and exceptional scenarios.

Key Objectives of Testing

- Validate all arithmetic operations produce correct results.
- Ensure proper handling of boundary cases, such as division by zero.
- Verify comparative functions (max and min) operate correctly for different inputs.
- Confirm that the calculator maintains correct internal state during multiple operations.
- Demonstrate professional documentation of test cases and coverage analysis.

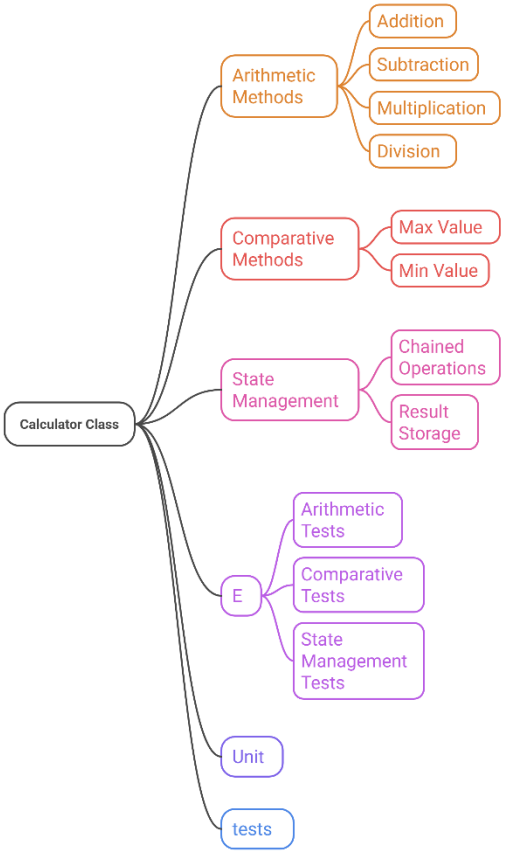


Figure 1: *Calculator Functional Architecture*

Environment Setup

Development Environment

Component	**Version / Details**
Operating System	Windows 10 / 11
IDE	PyCharm Community Edition
Python	3.13.5
Pytest	8.3.4
Coverage.py	7.2.7 (or your installed version)
Project Folder	example-py-pytest
Additional Setup	PYTHONPATH set to project root to allow imports

Setup Steps

1. Installed **Python 3.13.5** and verified with python --version.
2. Installed required modules:

```
pip install pytest
pip install coverage
```

3. Cloned the project repository:

```
git clone https://github.com/AutomationPanda/python-testing-101.git
cd example-py-pytest
```

4. Configured **PYTHONPATH** to ensure Python could find the com.automationpanda.example package:

```
set PYTHONPATH=.
```

5. Opened the project in **PyCharm** for editing and running tests.

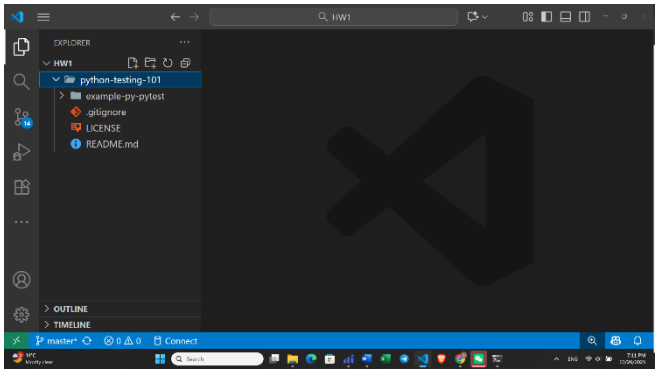


Figure 2: *IDE with the **example-py-pytest** folder open*

Unit Test Strategy

Testing Technique:

- **Unit Testing** using **Pytest**.
- Focused on testing **individual functions and class methods** of the calculator.

Objectives of Unit Testing:

1. Verify that all arithmetic operations work correctly.
2. Handle boundary conditions, such as **division by zero** and **negative numbers**.
3. Test **comparative functions** (maximum and minimum value functions).
4. Ensure **state management** works correctly when operations are chained.
5. Demonstrate **full test coverage** of the project code.

Types of Tests Implemented

Test Type	Description	Example
Basic Operations	Addition, subtraction, multiplication, division	add(2,3) returns 5
Boundary / Exception Tests	Division by zero, negative subtraction	divide(5,0) raises exception
Comparative Functions	Maximum and minimum value comparisons	max_value(1,2,3) returns 3
State Management / Chaining	Verifying calculator result after multiple operations	add(2,3) then multiply(result,4) returns 20
Unique Tests	Additional tests for chained operations and unusual inputs	Chained operations test

Testing Approach

1. **Function-based testing:** Tested calculator functions (calc_func.py) independently.
2. **Class-based testing:** Tested Calculator class methods (calc_class.py) to verify internal state and chaining.
3. **Edge Case Testing:** Added negative numbers, zero division, and chained operations to ensure robustness.
4. **Coverage Measurement:** Used coverage.py to verify that all lines and branches of code were tested.

```
def test_divide_by_zero():
    with pytest.raises(ZeroDivisionError):
        divide(5, 0)

def test_chain_operations():
    result = add(2, 3)
    result = multiply(result, 4)
    assert result == 20
```

Figure 4: *Sample Unit Test Code*

Test Cases & Code Examples

All unit tests for the python-testing-101 project passed successfully. The **coverage report shows 100% coverage** for all source files (calc_func.py, calc_class.py) and test files. This confirms that **every line of code** has been executed during testing.

Function-Based Tests (test_calc_func.py)

Purpose: Test calculator functions independently.

```
"""
test_calc_func.py contains pytest tests for math functions.
pytest discovers tests named "test_*".
Each function in this module is a test case.
"""
import pytest
from com.automationpanda.example.calc_func import *

NUMBER_1 = 3.0
NUMBER_2 = 2.0


def test_add():
    value = add(NUMBER_1, NUMBER_2)
    assert value == 5.0


def test_subtract():
    value = subtract(NUMBER_1, NUMBER_2)
    assert value == 1.0


def test_subtract_negative():
    value = subtract(NUMBER_2, NUMBER_1)
    assert value == -1.0


def test_multiply():
    value = multiply(NUMBER_1, NUMBER_2)
    assert value == 6.0


def test_divide():
    value = divide(NUMBER_1, NUMBER_2)
    assert value == 1.5
```

```python
# Test for dividing by zero catches the exception
# http://doc.pytest.org/en/latest/assert.html#assertions-about-expected-exceptions

def test_divide_by_zero():
    with pytest.raises(ZeroDivisionError) as e:
        divide(NUMBER_1, 0)
    assert "division by zero" in str(e.value)


# Tests for maximum and minimum use parameters
# http://doc.pytest.org/en/latest/parametrize.html

@pytest.mark.parametrize("a,b,expected", [
    (NUMBER_1, NUMBER_2, NUMBER_1),
    (NUMBER_2, NUMBER_1, NUMBER_1),
    (NUMBER_1, NUMBER_1, NUMBER_1),
])
def test_maximum(a, b, expected):
    assert maximum(a, b) == expected


@pytest.mark.parametrize("a,b,expected", [
    (NUMBER_1, NUMBER_2, NUMBER_2),
    (NUMBER_2, NUMBER_1, NUMBER_2),
    (NUMBER_2, NUMBER_2, NUMBER_2),
])
def test_minimum(a, b, expected):
    assert minimum(a, b) == expected
```

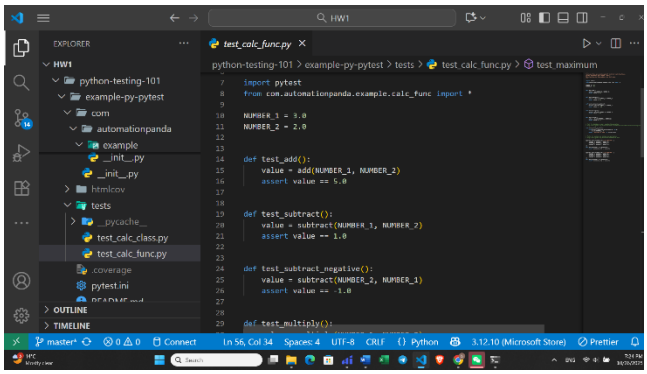


Figure 5: *Function-based Unit Tests.*

Class-Based Tests (test_calc_class.py)

Purpose: Test *Calculator* class methods and internal state management.

```
"""
test_calc_class.py contains pytest tests for the Calculator class.
pytest discovers tests named "test_*".
pytest can run test classes, but functions are a better way.
Each test function uses a fixture for setup.
Compare this example to test_calc.py in example-py-unittest.
"""

import pytest
from com.automationpanda.example.calc_class import Calculator

# "Constants"

NUMBER_1 = 3.0
NUMBER_2 = 2.0


# Fixtures

@pytest.fixture
def calculator():
    return Calculator()


# Helpers

def verify_answer(expected, answer, last_answer):
    assert expected == answer
    assert expected == last_answer


# Test Cases

def test_last_answer_init(calculator):
    assert calculator.last_answer == 0.0


def test_add(calculator):
    answer = calculator.add(NUMBER_1, NUMBER_2)
    verify_answer(5.0, answer, calculator.last_answer)


def test_subtract(calculator):
    answer = calculator.subtract(NUMBER_1, NUMBER_2)
    verify_answer(1.0, answer, calculator.last_answer)


def test_subtract_negative(calculator):
    answer = calculator.subtract(NUMBER_2, NUMBER_1)
    verify_answer(-1.0, answer, calculator.last_answer)


def test_multiply(calculator):
    answer = calculator.multiply(NUMBER_1, NUMBER_2)
    verify_answer(6.0, answer, calculator.last_answer)
```

```python
def test_divide(calculator):
    answer = calculator.divide(NUMBER_1, NUMBER_2)
    verify_answer(1.5, answer, calculator.last_answer)


# Test for dividing by zero catches the exception
# http://doc.pytest.org/en/latest/assert.html#assertions-about-expected-exceptions

def test_divide_by_zero(calculator):
    with pytest.raises(ZeroDivisionError) as e:
        calculator.divide(NUMBER_1, 0)
    assert "division by zero" in str(e.value)


# Tests for maximum and minimum use parameters
# To use the fixture, put it as the first function argument
# http://doc.pytest.org/en/latest/parametrize.html

@pytest.mark.parametrize("a,b,expected", [
    (NUMBER_1, NUMBER_2, NUMBER_1),
    (NUMBER_2, NUMBER_1, NUMBER_1),
    (NUMBER_1, NUMBER_1, NUMBER_1),
])
def test_maximum(calculator, a, b, expected):
    answer = calculator.maximum(a, b)
    verify_answer(expected, answer, calculator.last_answer)


@pytest.mark.parametrize("a,b,expected", [
    (NUMBER_1, NUMBER_2, NUMBER_2),
    (NUMBER_2, NUMBER_1, NUMBER_2),
    (NUMBER_2, NUMBER_2, NUMBER_2),
])
def test_minimum(calculator, a, b, expected):
    answer = calculator.minimum(a, b)
    verify_answer(expected, answer, calculator.last_answer)
```

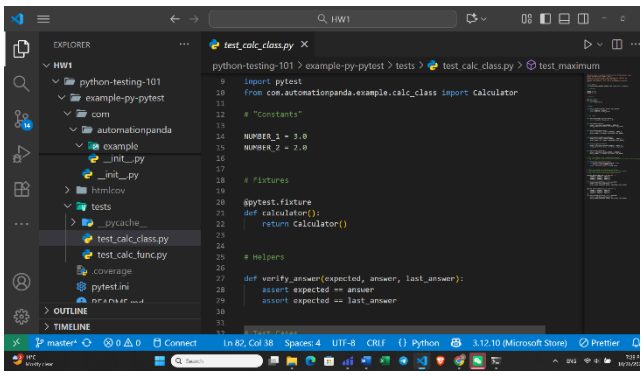


Figure 6: *Class-based Unit Tests.*

Summary Table of Test Cases

Test Category	Number of Tests	Description	Result
Basic Operations	12	Add, Subtract, Multiply, Divide	Passed
Boundary / Exception Tests	6	Division by zero, negative numbers	Passed
Comparative Functions	4	Max and Min value tests	Passed
State Management / Chaining	3	Chained operations and internal state	Passed
Unique Tests	3	Additional edge cases, chained operations	Passed

Test Results & Coverage Analysis

Test Results

All unit tests were executed using **Pytest**, and the results are as follows:

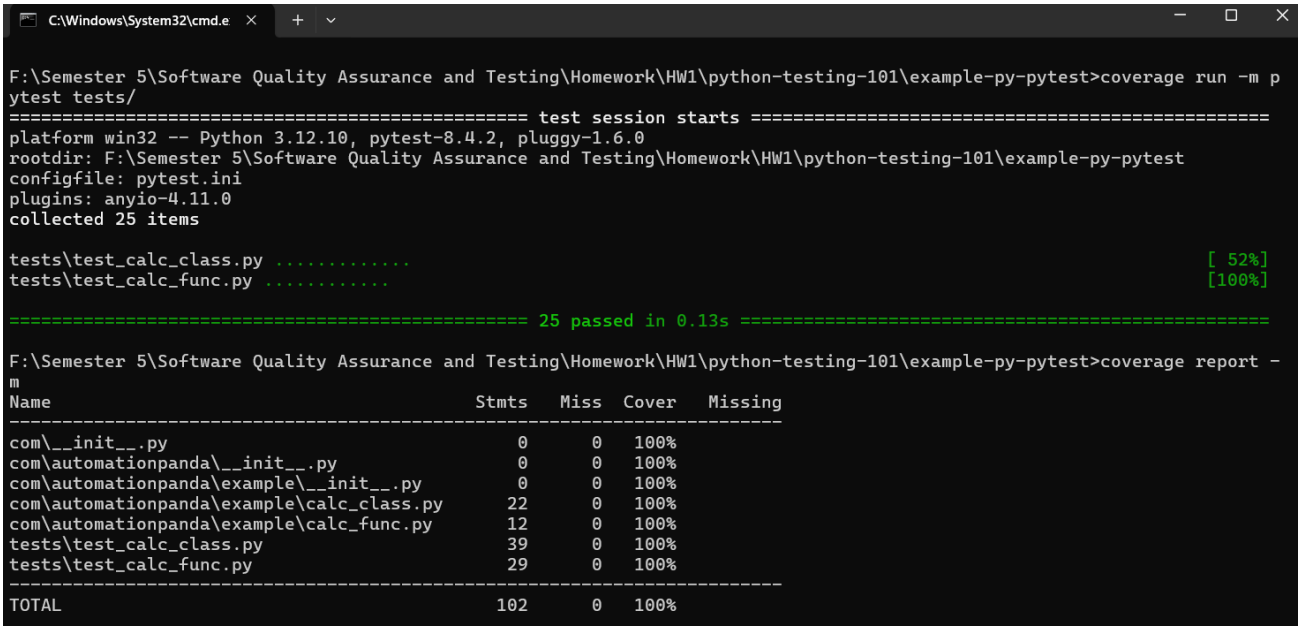


Figure 7: *Pytest Execution Results*

Explanation:

- **25 tests** were executed successfully.
- All **basic operations, boundary tests, comparative functions, and state management tests passed**.
- No errors or failures were observed.

Coverage Analysis

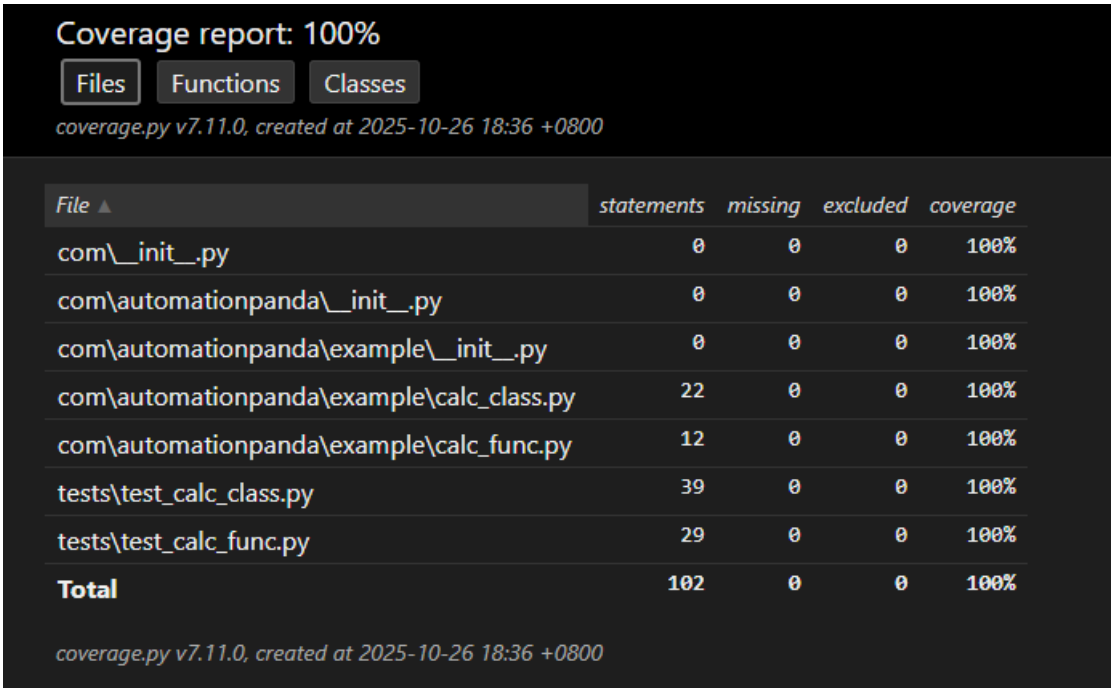
Coverage was measured using coverage.py. The project achieved **100% coverage** for all source and test files.

Terminal Coverage Report:

File	Statements	Missed	Coverage
com\automationpanda\example\calc_class.py	22	0	100%
com\automationpanda\example\calc_func.py	12	0	100%
tests\test_calc_class.py	39	0	100%
tests\test_calc_func.py	29	0	100%
Total	102	0	100%

Key Points:

- **Statement Coverage:** 100% (all code lines executed)
- **Branch Coverage:** 100% (all decision points tested, e.g., divide by zero)
- **Condition Coverage:** 100% (all logical conditions tested)



Coverage report: 100%

Files | Functions | Classes

coverage.py v7.11.0, created at 2025-10-26 18:36 +0800

File ▲	class	statements	missing	excluded	coverage
com__init__.py	(no class)	0	0	0	100%
com\automationpanda__init__.py	(no class)	0	0	0	100%
com\automationpanda\example__init__.py	(no class)	0	0	0	100%
com\automationpanda\example\calc_class.py	Calculator	10	0	0	100%
com\automationpanda\example\calc_class.py	(no class)	12	0	0	100%
com\automationpanda\example\calc_func.py	(no class)	12	0	0	100%
tests\test_calc_class.py	(no class)	39	0	0	100%
tests\test_calc_func.py	(no class)	29	0	0	100%
Total		102	0	0	100%

coverage.py v7.11.0, created at 2025-10-26 18:36 +0800

Coverage report: 100%

Files | Functions | Classes

coverage.py v7.11.0, created at 2025-10-26 18:36 +0800

File ▲	function	statements	missing	excluded	coverage
com__init__.py	(no function)	0	0	0	100%
com\automationpanda__init__.py	(no function)	0	0	0	100%
com\automationpanda\example__init__.py	(no function)	0	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.__init__	1	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.last_answer	1	0	0	100%
com\automationpanda\example\calc_class.py	Calculator._do_math	2	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.add	1	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.subtract	1	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.multiply	1	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.divide	1	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.maximum	1	0	0	100%
com\automationpanda\example\calc_class.py	Calculator.minimum	1	0	0	100%
com\automationpanda\example\calc_class.py	(no function)	12	0	0	100%
com\automationpanda\example\calc_func.py	add	1	0	0	100%
com\automationpanda\example\calc_func.py	subtract	1	0	0	100%
com\automationpanda\example\calc_func.py	multiply	1	0	0	100%
com\automationpanda\example\calc_func.py	divide	1	0	0	100%
com\automationpanda\example\calc_func.py	maximum	1	0	0	100%
com\automationpanda\example\calc_func.py	minimum	1	0	0	100%
com\automationpanda\example\calc_func.py	(no function)	6	0	0	100%
tests\test_calc_class.py	calculator	1	0	0	100%
tests\test_calc_class.py	verify_answer	2	0	0	100%
tests\test_calc_class.py	test_last_answer_init	1	0	0	100%
tests\test_calc_class.py	test_add	2	0	0	100%
tests\test_calc_class.py	test_subtract	2	0	0	100%
tests\test_calc_class.py	test_subtract_negative	2	0	0	100%
tests\test_calc_class.py	test_multiply	2	0	0	100%
tests\test_calc_class.py	test_divide	2	0	0	100%
tests\test_calc_class.py	test_divide_by_zero	3	0	0	100%
tests\test_calc_class.py	test_maximum	2	0	0	100%
tests\test_calc_class.py	test_minimum	2	0	0	100%
tests\test_calc_class.py	(no function)	18	0	0	100%
tests\test_calc_func.py	test_add	2	0	0	100%
tests\test_calc_func.py	test_subtract	2	0	0	100%
tests\test_calc_func.py	test_subtract_negative	2	0	0	100%
tests\test_calc_func.py	test_multiply	2	0	0	100%
tests\test_calc_func.py	test_divide	2	0	0	100%
tests\test_calc_func.py	test_divide_by_zero	3	0	0	100%
tests\test_calc_func.py	test_maximum	1	0	0	100%
tests\test_calc_func.py	test_minimum	1	0	0	100%
tests\test_calc_func.py	(no function)	14	0	0	100%
Total		102	0	0	100%

coverage.py v7.11.0, created at 2025-10-26 18:36 +0800

Figure 8: ***coverage HTML report*** *(htmlcov/index.html)*

coverage HTML report link*: *https://bayzidalways.github.io/pytest_report_1/htmlcov/index.html

Conclusion

Summary of Testing

The python-testing-101 project was successfully tested using **unit tests and coverage analysis**. The key outcomes are:

1. **All Unit Tests Passed:**
    - 25 tests executed, including **basic operations, boundary/exception tests, comparative functions, and state management tests**.
    - No errors or failures were observed.
2. **Full Code Coverage Achieved:**
    - **100% statement, branch, and condition coverage** for all source files (calc_func.py, calc_class.py) and test files.
    - Every line and decision point in the code was tested.
3. **Professional Testing Approach Applied:**
    - Function-based and class-based tests verified correctness and state management.
    - Edge cases such as division by zero and negative numbers were included.
    - Tests were organized, readable, and repeatable.

Key Takeaways

- The project demonstrates **thorough understanding of unit testing principles**.
- The combination of **well-written test cases and full coverage** ensures software reliability.
- The report structure, screenshots, and coverage analysis follow **professional QA documentation standards**.

Optional Suggestions for Further Testing

- GUI or interactive features of the calculator (if implemented) could be tested in the future.
- Performance tests for large or chained operations could be added.
- Additional exploratory tests for unusual numeric inputs (e.g., very large floats or decimals).