Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Кафедра оптоелектроніки та інформаційних технологій

Звіт
про виконання лабораторної роботи №1
на тему:

«Перевірка результату виконання програми за допомогою юніт тестування»

Виконала:
студентка групи ФеС-32
Філь Дарина

Львів - 2025

Мета:

Ознайомитися з методологією юніт тестування, навчитися створювати та виконувати юніт тести для перевірки правильності роботи програмного коду.

Теоретичні відомості:

Юніт тестування (Unit Testing) - це метод тестування програмного забезпечення, який передбачає перевірку окремих модулів програми (функцій, методів) для підтвердження їхньої коректної роботи.

Юніт-тести допомагають знайти помилки ще на ранніх етапах розробки, що спрощує подальше підтримання та розвиток коду.

Основні принципи юніт тестування:

1. **Автоматизованість** - тести виконуються автоматично без втручання користувача.
2. **Ізольованість** - кожен тест перевіряє лише один конкретний аспект коду, тести не залежать від інших частин системи, баз даних або API.
3. **Повторюваність** - тести можуть виконуватися багаторазово з однаковими результатами.
4. **Незалежність** - тести не повинні залежати один від одного.
5. **Мале охоплення** - тестують окремі функції або класи.

Юніт тестування використовується у багатьох мовах програмування. Приклади популярних фреймворків:

- Python - unittest, pytest
- Java - JUnit
- C# - NUnit
- JavaScript - Jest, Mocha
- Ruby - RSpec

Для написання юніт тестів у Python зазвичай використовується бібліотека `unittest`. Вона дозволяє створювати тести, виконувати їх і перевіряти результати.

Завдання до роботи:

1. Реалізувати функцію, яка розв’язує квадратне рівняння.
2. Створити набір юніт тестів для перевірки правильності роботи цієї функції.
3. Реалізувати мок-об'єкт для даної функції.
4. Виконати завдання з використанням бібліотеки unittest, а також додатково — з використанням довільної іншої бібліотеки для юніт тестування.
5. Запустити тести та проаналізувати їх результат.
6. Навмисно внести помилку у функцію.
7. Повторно запустити тести та проаналізувати результат.

Хід роботи

1. Реалізувати функцію, яка розв’язує квадратне рівняння.

```python
def quadratic(a, b, c):  7 usages
    d = b ** 2 - 4 * a * c

    if d < 0:
        return 'Result is complex'

    x1 = (-b + sqrt(d)) / (2 * a)
    x2 = (-b - sqrt(d)) / (2 * a)

    return (x1,) if d == 0 else (x1, x2)
```

2. Створити набір юніт тестів для перевірки правильності роботи цієї функції

```python
class TestQuadratic(TestCase):
    def test_complex_roots(self):
        self.assertEqual(quadratic( a: 1,  b: 2,  c: 3),  second: 'Result is complex')

    def test_one_root(self):
        self.assertEqual(quadratic( a: 1, -4,  c: 4),  second: (2.0,))

    def test_two_roots(self):
        self.assertEqual(quadratic( a: 1, -6,  c: 5),  second: (5.0, 1.0))

    def test_zero_division(self):
        with self.assertRaises(ZeroDivisionError):
            quadratic( a: 0,  b: 1,  c: 2)
```

3. Реалізувати мок-об'єкт для даної функції.

```
quadratic_mock = Mock()
quadratic_mock.return_value = (2.0, -0.5)

def test_mock():  1 usage
    assert quadratic_mock( *args: 2, -3, -2) == (2.0, -0.5)
    quadratic_mock.assert_called_with( *args: 2, -3, -2)
```

4. Виконати завдання з використанням бібліотеки unittest, а також додатково — з використанням довільної іншої бібліотеки для юніт тестування.

```
def test_assert_complex_roots():
    assert quadratic( a: 1,  b: 2,  c: 3) == 'Result is complex'

def test_assert_one_root():
    assert quadratic( a: 1, -4,  c: 4) == (2.0,)

def test_assert_two_roots():
    assert quadratic( a: 1, -6,  c: 5) == (5.0, 1.0)
```

5. Запустити тести та проаналізувати їх результат.

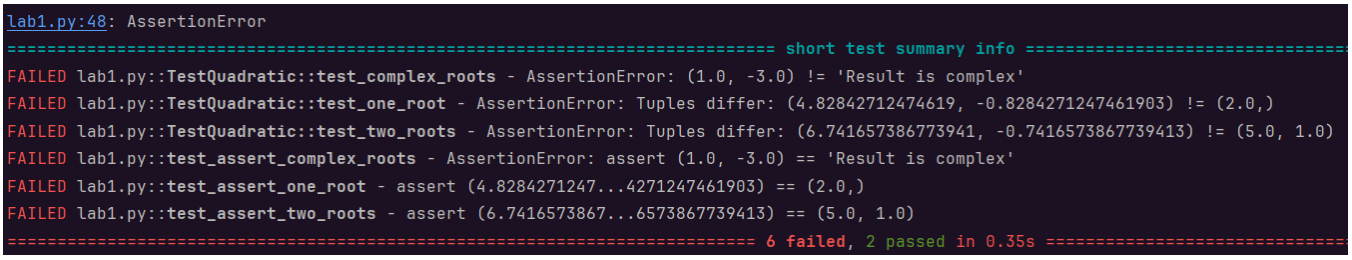
```
PS E:\СПП\Lab1> pytest lab1.py
===== test session starts ==========================================================================
platform win32 -- Python 3.12.8, pytest-8.3.5, pluggy-1.5.0
rootdir: E:\СПП\Lab1
collected 8 items

lab1.py ........

========================================================================== 8 passed in 0.15s
```

6. Навмисно внести помилку у функцію.

```
def quadratic(a, b, c):  7 usages
    d = b ** 2 - 4 * a * c
```

```
def quadratic(a, b, c):  7 usages
    d = b ** 2 + 4 * a * c
```

7. Повторно запустити тести та проаналізувати результат.



Висновок:

У цій лабораторній роботі я ознайомилася з методологією юніт тестування, навчилася створювати та виконувати юніт тести для перевірки правильності роботи програмного коду. Також я створила мок-об’єкт та провела тестування з використанням бібліотеки unittest та pytest. Під час виконання було виявлено, що тести дозволяють ефективно перевірити коректність функції та своєчасно виявити помилки.

Додаток

Код програми:

```
from math import sqrt
from unittest import TestCase, main
from unittest.mock import Mock

def quadratic(a, b, c):
    d = b ** 2 - 4 * a * c

    if d < 0:
        return 'Result is complex'

    x1 = (-b + sqrt(d)) / (2 * a)
    x2 = (-b - sqrt(d)) / (2 * a)

    return (x1,) if d == 0 else (x1, x2)
```

```
class TestQuadratic(TestCase):
    def test_complex_roots(self):
        self.assertEqual(quadratic(1, 2, 3), 'Result is complex')

    def test_one_root(self):
        self.assertEqual(quadratic(1, -4, 4), (2.0,))

    def test_two_roots(self):
        self.assertEqual(quadratic(1, -6, 5), (5.0, 1.0))

    def test_zero_division(self):
        with self.assertRaises(ZeroDivisionError):
            quadratic(0, 1, 2)

quadratic_mock = Mock()
quadratic_mock.return_value = (2.0, -0.5)

def test_mock():
    assert quadratic_mock(2, -3, -2) == (2.0, -0.5)
    quadratic_mock.assert_called_with(2, -3, -2)

def test_assert_complex_roots():
    assert quadratic(1, 2, 3) == 'Result is complex'

def test_assert_one_root():
    assert quadratic(1, -4, 4) == (2.0,)

def test_assert_two_roots():
    assert quadratic(1, -6, 5) == (5.0, 1.0)

if __name__ == '__main__':
    test_mock()
    main()
```