

Санкт-Петербургский Национальный Исследовательский Университет
Информационных Технологий, Механики и Оптики

Факультет Инфокоммуникационных Технологий

Тестирование программного обеспечения

Лабораторная работа 1

Выполнил:

Колсанов Я. И.

Проверил:

Кочубеев Н. С.

Санкт-Петербург, 2024

Цель: научиться писать unit тесты для существующего проекта.

Задачи:

1. Выбрать открытый проект на платформе GitHub.
2. Проанализировать функциональность приложения и определить, какие модули или компоненты требуют тестирования. При этом выделить функциональные элементы, критические части системы, которые должны быть протестированы, и важные случаи использования (use cases).
3. Создать модульные тесты для выбранных компонентов системы с использованием AAA (arrange, act, assert) и FIRST (fast, isolated, repeatable, self-validating, timely) principles.

Ход работы

Ссылка на репозиторий с тестами:

<https://github.com/yaroslavkolsanov/TestPO>

1. Выбор проекта на GitHub

Был выбран следующий репозиторий:

<https://github.com/GluKhovKirill/Great-Calculator>

Он представляет из себя небольшой проект графического калькулятора с несколькими файлами, написанный на языке Python. Основная логика и функционал приложения содержатся в файле `logic.py`, а именно в методах класса `MathExecutor`, каждый из которых выполняет определенную математическую операцию. Unit тесты для проекта также написаны на Python.

2. Анализ тестируемых функциональностей

Функциональные элементы:

- Арифметические операции: сложение (`amount`), вычитание (`difference`), умножение (`multiply`), деление (`division`), возведение в степень (`raise_to_a_power`), вычисление факториала (`factorial`).
- Тригонометрические функции: синус (`sine`), косинус (`cosine`), тангенс (`tangent`), котангенс (`cotangent`).
- Операции с числами: извлечение корней (квадратного и кубического), модуль числа (`module`), инверсия числа по знаку (`invert_number`).
- Работа с константами: π (`pi`), e (`e`).
- Логарифмические операции: логарифм числа (`log`).
- Перевод градусов в радианы и наоборот (`degrees_to_radians`, `radians_to_degrees`).
- Перевод в проценты и обратно (`percents`, `un_pcents`).
- Перевод числа в десятичную систему из любой другой (`in_decimal_system`).

Критические части системы:

- Обработка арифметических ошибок: деление на ноль, ввод нечисловых значений, ввод отрицательных значений для операций, где они недопустимы (например, извлечение квадратного корня).
- Тригонометрические функции: синус, косинус и тангенс, особенно с учётом перевода углов из градусов в радианы и обратно.
- Функции, работающие с процентами и системами счисления: правильность конвертации данных должна быть проверена.
- Обработка исключений: система должна корректно обрабатывать некорректные входные данные (например, строки вместо чисел).

Важные случаи использования (use cases):

- Базовые арифметические операции: сложение двух чисел, вычитание, умножение, деление. Граничный случай - деление на ноль.
- Вычисление синуса, косинуса и тангенса для углов в градусах и радианах. Граничный случай - углы 0° , 90° , 180° , 360° .
- Факториал для положительных целых чисел. Граничный случай - факториал 0 (результат должен быть 1). Ошибочные случаи - ввод нецелого числа (должна быть ошибка).
- Перевод десятичного числа в проценты и обратно.
- Возведение числа в положительную и отрицательную степень. Граничный случай — возведение в нулевую степень (результат должен быть 1).

3. Написание тестов

Далее были написаны тесты для основных функций приложения.

```
class TestDivision(unittest.TestCase):
    """Тест 1: Проверка операции деления"""

    def test_zero_division(self):
        executor = MathExecutor(1, ":", 0, False) # Деление на ноль
        result = executor.execute()
        self.assertEqual(result, "Вы попытались поделить на 0. К сожалению, это ещё не определено!")

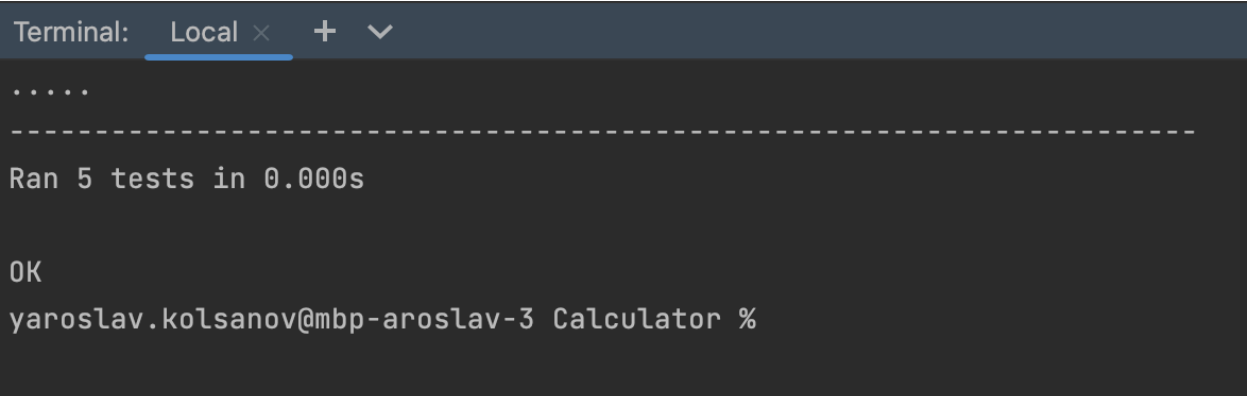
    def test_str_division(self):
        executor = MathExecutor(1, ":", 'Два', False) # Строка вместо числа
        result = executor.execute()
        self.assertEqual(result, "Операнд не является числом")

    def test_base_division(self):
        executor = MathExecutor(2, ":", 1, False) # Базовое деление
        result = executor.execute()
        self.assertEqual(result, "2.0")

    def test_float_division(self):
        executor = MathExecutor(2, ":", 0.5, False) # Деление на десятичную дробь
        result = executor.execute()
        self.assertEqual(result, "4.0")

    def test_neg_division(self):
        executor = MathExecutor(10, ":", -5, False) # Деление на отрицательное число
        result = executor.execute()
        self.assertEqual(result, "-2.0")
```

Рисунок 1 – Тестирование функции, выполняющей операцию деления



```
Terminal: Local x + v
.....
-----
Ran 5 tests in 0.000s

OK
yaroslav.kolsanov@mbp-aroslav-3 Calculator %
```

Рисунок 2 – Результат

```

class TestTrigonometric(unittest.TestCase):
    """Тест 2: Проверка тригонометрических функций"""

    def test_not_defined_value(self):
        executor = MathExecutor(90, "tg", None, True) # Тангенс 90 градусов не определен
        result = executor.execute()
        self.assertEqual(result, "Тангенс не определен")

    def test_str_value(self):
        executor = MathExecutor("Угол", "ctg", None, True) # Строка вместо числа
        result = executor.execute()
        self.assertEqual(result, "Операнд не является числом")

    def test_neg_degree(self):
        executor = MathExecutor(-30, "cos", None, True) # Косинус отрицательного угла
        result = executor.execute()
        self.assertEqual(round(float(result), 3), round(3 ** 0.5 / 2, 3))

    def test_base_func_value(self):
        executor = MathExecutor(30, "sin", None, True) # Синус обычного угла
        result = executor.execute()
        self.assertEqual(round(float(result), 1), 0.5)

    def test_period_func_value(self):
        executor_1 = MathExecutor(30, "cos", None, True) # Косинус обычного угла
        executor_2 = MathExecutor(30 + 360, "cos", None, True) # Косинус обычного угла с периодом 2pi
        result_1 = executor_1.execute()
        result_2 = executor_2.execute()
        self.assertEqual(round(float(result_1), 3), round(float(result_2), 3))

```

Рисунок 3 – Тестирование тригонометрических функций

```

Terminal: Local x + v
=====
FAIL: test_not_defined_value (test_calculator.TestTrigonometric)
-----
Traceback (most recent call last):
  File "/Users/yaroslav.kolsanov/PycharmProjects/Calculator/test_calculator.py", line 40, in test_not_defined_value
    self.assertEqual(result, "Тангенс не определен")
AssertionError: '1.633123935319537e+16' != 'Тангенс не определен'
- 1.633123935319537e+16
+ Тангенс не определен

-----
Ran 5 tests in 0.001s

FAILED (failures=1, errors=1)
yaroslav.kolsanov@mbp-aroslav-3 Calculator %

```

Рисунок 4 – Результат

Возникла 1 неудача, так как значение тангенса 90 градусов не определено, что не предусмотрено в исходном коде. Также возникла 1 ошибка, связанная с тем, что в приложении нет обработки в случае ввода символов вместо числа.

```

class TestFactorial(unittest.TestCase):
    """Тест 3: Проверка факториала числа"""

    def test_factorial_of_null(self):
        executor = MathExecutor(0, "n!", None, False) # Факториал нуля
        result = executor.execute()
        self.assertEqual(result, "1")

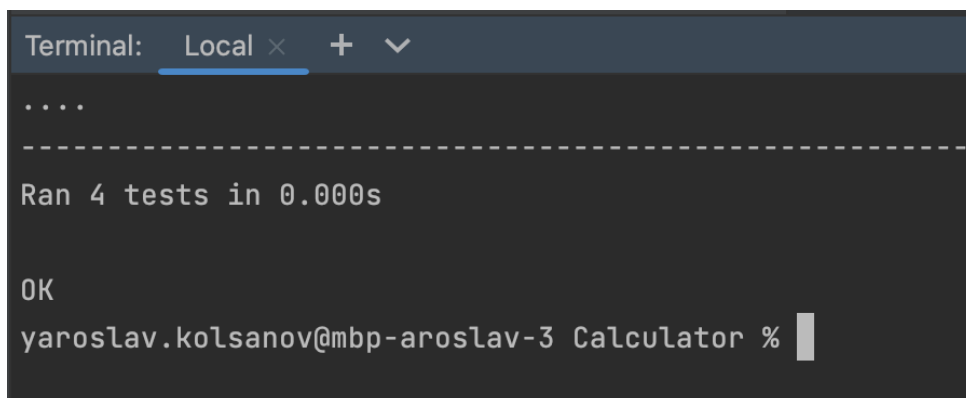
    def test_factorial_of_str_value(self):
        executor = MathExecutor("Число", "n!", None, True) # Строка вместо числа
        result = executor.execute()
        self.assertEqual(result, "Операнд не является целочисленным числом (integer)")

    def test_factorial_of_neg(self):
        executor = MathExecutor(-5, "n!", None, False) # Факториал отрицательного числа
        result = executor.execute()
        self.assertEqual(result, "Операнд не является целочисленным числом (integer)")

    def test_base_factorial(self):
        executor = MathExecutor(5, "n!", None, False) # Факториал натурального числа
        result = executor.execute()
        self.assertEqual(result, "120")

```

Рисунок 5 – Тестирование функции нахождения факториала числа



The image shows a terminal window with a dark background. At the top, there's a title bar with 'Terminal:' and a tab labeled 'Local' with a close button (x) and window controls (+ and v). Below the title bar, the terminal output shows four dots '....' followed by a dashed line separator. Under the separator, it says 'Ran 4 tests in 0.000s'. Below that, it says 'OK'. At the bottom, the prompt 'yaroslav.kolsanov@mbp-aroslav-3 Calculator %' is visible with a cursor at the end.

```

Terminal: Local x + v
....
-----
Ran 4 tests in 0.000s

OK
yaroslav.kolsanov@mbp-aroslav-3 Calculator %

```

Рисунок 6 – Результат

```

class TestPercents(unittest.TestCase):
    """Тест 4: Проверка перевода числа в проценты"""

    def test_str_value_percents(self):
        executor = MathExecutor("Процент", "%", None, False) # Строка вместо числа
        result = executor.execute()
        self.assertEqual(result, "Операнд не является числом")

    def test_float_value_percents(self):
        executor = MathExecutor(0.5, "%", None, False) # Десятичная дробь
        result = executor.execute()
        self.assertEqual(result, "50.0%")

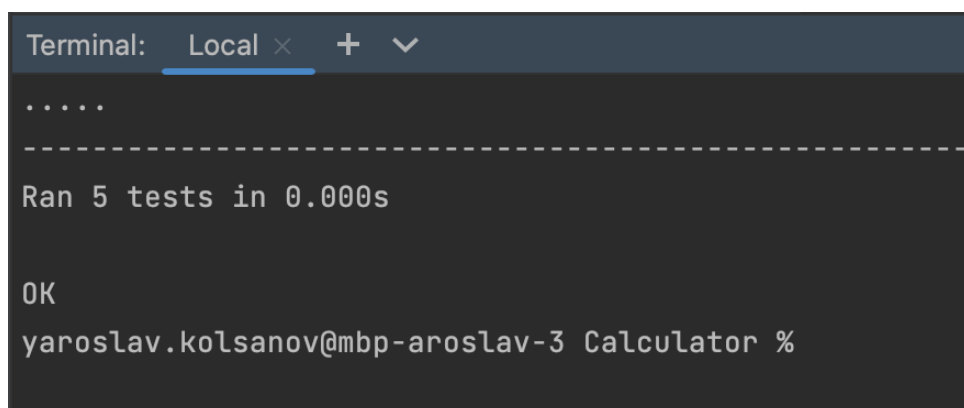
    def test_natural_value_percents(self):
        executor = MathExecutor(10, "%", None, False) # Обычное число
        result = executor.execute()
        self.assertEqual(result, "1000.0%")

    def test_null_value_percents(self):
        executor_3 = MathExecutor(0, "%", None, False) # Ноль
        result_3 = executor_3.execute()
        self.assertEqual(result_3, "0.0%")

    def test_neg_value_percents(self):
        executor_4 = MathExecutor(-0.5, "%", None, False) # Отрицательное число
        result_4 = executor_4.execute()
        self.assertEqual(result_4, "-50.0%")

```

Рисунок 7 – Тестирование функции перевода числа в проценты



The image shows a terminal window with a dark background. At the top, there's a title bar that says "Terminal: Local" followed by a close button (x) and a dropdown arrow (v). Below the title bar, the text "....." is displayed, followed by a dashed line. Underneath the dashed line, it says "Ran 5 tests in 0.000s". Below that, the text "OK" is shown. At the bottom of the terminal, the prompt "yaroslav.kolsanov@mbp-aroslav-3 Calculator %" is visible.

```

Terminal: Local x v
.....
-----
Ran 5 tests in 0.000s

OK
yaroslav.kolsanov@mbp-aroslav-3 Calculator %

```

Рисунок 8 – Результат


```

class TestDecimalSystem(unittest.TestCase):
    """Тест 5: Проверка перевода числа в десятичную систему счисления"""

    def test_str_value_percents(self):
        executor = MathExecutor("Двоичное число", "n to10", 2, False) # Строка вместо числа
        result = executor.execute()
        self.assertEqual(result, "Операнд не является целочисленным числом (integer) "
                               "или такое число невозможно в данной системе")

    def test_natural_value_from_binary_to_decimal_system(self):
        executor = MathExecutor(10, "n to10", 2, False) # Обычное двоичное число
        result = executor.execute()
        self.assertEqual(result, "2")

    def test_natural_value_from_decimal_to_binary_system(self):
        executor = MathExecutor(10, "n to10", 10, False) # Десятичное число
        result = executor.execute()
        self.assertEqual(result, "10")

    def test_neg_value_from_binary_to_decimal_system(self):
        executor_3 = MathExecutor(-10, "n to10", 2, False) # Отрицательное двоичное число
        result_3 = executor_3.execute()
        self.assertEqual(result_3, "-2")

    def test_float_value_from_binary_to_decimal_system(self):
        executor_4 = MathExecutor(10.11, "n to10", 2, False) # Двоичное дробное число
        result_4 = executor_4.execute()
        self.assertEqual(result_4, "2.75")

```

Рисунок 9 – Тестирование функции перевода числа в десятичную систему счисления

```

Terminal: Local x + v
=====
FAIL: test_float_value_from_binary_to_decimal_system (test_calculator.TestDecimalSystem)
=====
Traceback (most recent call last):
  File "/Users/yaroslav.kolsanov/PycharmProjects/Calculator/test_calculator.py", line 145, in test_float_value_from_binary_to_decimal_system
    self.assertEqual(result_4, "2.75")
AssertionError: 'Операнд не является целочисленным числом [49 chars]теме' != '2.75'
- Операнд не является целочисленным числом (integer) или такое число невозможно в данной системе
+ 2.75

-----
Ran 5 tests in 0.000s

FAILED (failures=1)
yaroslav.kolsanov@mbp-aroslav-3 Calculator %

```

Рисунок 10 – Результат

Возникла 1 неудача, так как в функции не предусмотрен ввод двоичной дроби, и в следствие чего, не предусмотрен ее перевод в десятичную систему счисления.

Вывод: в результате выполнения лабораторной работы был проведен анализ функциональности выбранного на GitHub проекта и определены модули и компоненты, требующие тестирования. Затем для них были написаны сами модульные тесты с использованием AAA и FIRST, при этом протестированы несколько сценариев работы, включая граничные случаи. Качество тестирования можно считать достаточным, так как было охвачено множество основных критических частей системы.