

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИТМО

ITMO University

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1
«Unit tests»

По дисциплине: Тестирование ПО

Обучающийся: Арройо Николас | Телеграм: [@nicarry](#)

Факультет: Факультет инфокоммуникационных технологий

Группа: K3320

Направление подготовки: 11.03.02 Инфокоммуникационные технологии и
системы связи

Образовательная программа: Программирование в
инфокоммуникационных системах

Обучающийся	_____	_____	_____
	(дата)	(подпись)	Арройо Н. (Ф.И.О.)

Руководитель	_____	_____	_____
	(дата)	(подпись)	Слюсаренко С. В. (Ф.И.О.)

Санкт Петербург

2025

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ	2
ВВЕДЕНИЕ	4
Рабочее окружение и организация кода:	4
Кратко о применяемых инструментах:	5
ОПИСАНИЕ ВЫБРАННОГО ПРОЕКТА И ЕГО КЛЮЧЕВЫХ МОДУЛЕЙ	6
КРАТКАЯ ХАРАКТЕРИСТИКА	6
КЛЮЧЕВЫЕ ФУНКЦИИ И НАЗНАЧЕНИЕ	6
1) <code>add(a, b)</code> — сложение.	6
2) <code>sub(a, b)</code> — вычитание.	6
3) <code>mul(a, b)</code> — умножение.	7
4) <code>div(a, b)</code> — деление.	7
5) <code>pow_int(base, exp)</code> — возведение в степень с целым показателем.	7
6) <code>factorial(n)</code> — факториал для целых $n \geq 0$.	8
Почему модуль подходит для юнит-тестирования	9
НАПИСАННЫЕ ТЕСТЫ: ПЕРЕЧЕНЬ, КРАТКИЕ ПОЯСНЕНИЯ И СВЯЗЬ С AAA/FIRST	9
Подход к тестированию	9
Тесты	10
1) <code>test_addition.py</code> — функция <code>add(a, b)</code>	10
2) <code>test_subtraction.py</code> — функция <code>sub(a, b)</code>	11
3) <code>test_multiplication.py</code> — функция <code>mul(a, b)</code>	12
4) <code>test_division.py</code> — функция <code>div(a, b)</code>	13
5) <code>test_pow_int.py</code> — функция <code>pow_int(base, exp)</code>	14
6) <code>test_factorial.py</code> — функция <code>factorial(n)</code>	15
7) <code>test_properties.py</code> — property-подход для <code>add/mul</code>	16
8) <code>test_randomized.py</code> — random-based testing	17
Результаты запуска тестов и метрику code coverage	18
Коррекция упавшего теста	19
Mutation testing	20
Добивание выживших мутантов (<code>pow_int</code>)	22
ЗАКЛЮЧЕНИЕ	24

ВВЕДЕНИЕ

Данная лабораторная работа посвящена разработке и запуску юнит-тесты для существующего проекта с применением различных инструментов, как `pytest`, `unittest` и `MutPy`.

Цель работы — показать полный цикл: анализ функциональности и выделение критичных участков, написание устойчивых и изолированных тестов по принципам AAA (Arrange–Act–Assert) и FIRST (Fast, Isolated, Repeatable, Self-validating, Timely), а также интерпретацию метрик качества — в первую очередь показателя покрытия кода (code coverage) и результатов мутационного тестирования.

В качестве тестируемого проекта выбран небольшой модуль [«calc»](#). Этот модуль был создан мной специально для данной лабораторной внутри [основного репозитория](#), поэтому здесь не используется fork или отдельный подрепозиторий: код приложения и тесты сосредоточены в одном месте для простоты проверки и воспроизводимости.

Рабочее окружение и организация кода:

- Лабораторная работа № 1 находится в репозитории [«Software-testing»](#).
- Структура каталога Lab1:
 - [Lab1/calc](#) — исходный код мини-проекта (модуль операций калькулятора)
 - [Lab1/tests](#) — мои юнит-тесты.
 - [Lab1/pytest.ini](#) и [Lab1/.coveragerc](#) — конфигурация запуска `pytest` и отчёта о покрытии.

- [Lab1/requirements-dev.txt](#) — dev-зависимости для локального запуска тестов.

Кратко о применяемых инструментах:

- `pytest` — основной test-runner (лаконичный синтаксис тестов и маркировок, удобный вывод).
 - Для сокращения дублирования и повышения вариативности входов применялся `data-driven testing` через `@pytest.mark.parametrize`.
 - Наборы данных покрывают положительные / отрицательные / десятичные значения и граничные случаи (например, деление на ноль, типовые ошибки параметров).
- `unittest` — дополнительно использован для демонстрации альтернативного стиля (класс `TestCase`).
- `pytest-cov` — сбор и репортинг показателя покрытия строк/ветвей.
- `MutPy` — мутационное тестирование; библиотека автоматически порождает “мутантов” (изменяет AST кода операторными мутациями, например ROR/AOR) и проверяет, “убивают” ли их тесты. Запуск интегрирован в GitHub Actions с публикацией отчётов (текст/HTML) как artifacts.

Описание выбранного проекта и его ключевых модулей

Краткая характеристика

Мини-проект [«calc»](#) — это небольшой модуль с «чистыми» арифметическими функциями, предназначенный для демонстрации принципов юнит-тестирования. Выбранная предметная область проста, но при этом позволяет показать тесты на корректные и граничные случаи, а также property-подход и мутационное тестирование.

Ключевые функции и назначение

1) `add(a, b)` — сложение.

```
def add(a: Number, b: Number) -> Number:  
    return a + b
```

Рисунок 1. Функция сложения

Назначение: вернуть сумму двух чисел.

Причина критичности: базовая арифметическая операция; используется в тестах свойств (коммутативность, дистрибутивность).

2) `sub(a, b)` — вычитание.

```
def sub(a: Number, b: Number) -> Number:  
    return a - b
```

Рисунок 2. Функция вычитания

Назначение: вернуть разность $a - b$.

Причина критичности: проверка корректности знаков в крайних случаях (отрицательный результат).

3) `mul(a, b)` — умножение.

```
def mul(a: Number, b: Number) -> Number:  
    return a * b
```

Рисунок 3. Функция умножения

Назначение: вернуть произведение двух чисел.

Причина критичности: краевые случаи (умножение на ноль, отрицательные множители), участие в property-тестах (коммутативность, дистрибутивность).

4) `div(a, b)` — деление.

```
def div(a: Number, b: Number) -> Number:  
    if b == 0:  
        raise ZeroDivisionError("division by zero")  
    return a / b
```

Рисунок 4. Функция деления

Назначение: вернуть частное a / b .

Валидация: при $b == 0$ выбрасывается `ZeroDivisionError`.

Причина критичности: классический крайний случай «деление на ноль», обязательный для тестов на ошибки.

5) `pow_int(base, exp)` — возведение в степень с целым показателем.

```
def pow_int(base: Number, exp: int) -> Number:  
    if not isinstance(exp, int):  
        raise TypeError("exp must be int")  
    if base == 0 and exp < 0:  
        raise ZeroDivisionError("0 cannot be raised to a negative power")  
    return base ** exp
```

Рисунок 5. Функция степени

Назначение: вернуть `base ** exp`; допускаются отрицательные показатели, кроме случая `base == 0` и `exp < 0` (`ZeroDivisionError`).

Валидация: `exp` должен быть `int`; при неверном типе — `TypeError`.

Причина критичности: демонстрация проверок параметров и работы с крайними случаями (ноль в отрицательной степени).

6) `factorial(n)` — факториал для целых $n \geq 0$.

```
def factorial(n: int) -> int:
    if not isinstance(n, int):
        raise TypeError("n must be int")
    if n < 0:
        raise ValueError("n must be >= 0")
    res = 1
    for k in range(2, n + 1):
        res *= k
    return res
```

Рисунок 6. Функция вычисления факториала

Назначение: вернуть $n!$; реализован итеративно.

Валидация: `n` должен быть `int`; `n < 0` — `ValueError`.

Причина критичности: классическая функция с чёткими пред- и пост-условиями; хорошо подходит для проверки ошибок и базовых случаев ($0!$, $1!$).

Почему модуль подходит для юнит-тестирования

- Все функции «чистые»: для данных входов всегда один и тот же выход; отсутствие побочных эффектов.
- Быстрые вычисления (мгновенный запуск тестов) и простые пред-/пост-условия, то есть легко формулировать AAA-тесты и граничные случаи.
- Наличие естественных алгебраических свойств (коммутативность сложения и умножения, дистрибутивность умножения относительно сложения), поэтому удобно показать property-подход без дополнительных библиотек.

Написанные тесты: перечень, краткие пояснения и связь с AAA/FIRST

Подход к тестированию

- Для сокращения дублирования и повышения вариативности входов применялся data-driven testing.
- В каждом тесте явно соблюдается AAA: Arrange (подготовка входных данных), Act (вызов тестируемой функции), Assert (проверка результата/исключения). Нотация AAA в самом коде тестов отмечена комментариями “Arrange / Act / Assert”
- FIRST:
 - Fast — все функции «чистые», тесты работают in-memory и выполняются очень быстро;

- Isolated — тесты не зависят от внешней среды и не делятся состоянием;
- Repeatable — результаты детерминированы для одинаковых входов;
- Self-validating — проверки выполнены строгими assert-ами;
- Timely — тесты написаны параллельно с анализом функциональности.

Тесты

1) [test_addition.py](#) — функция add(a, b)

```
import pytest
from calc.operations import add

    # Arrange
@pytest.mark.parametrize("a,b,expected",
    (2, 3, 5),
    (0, 0, 0),
    (-1, 1, 0),
    (2.5, 0.5, 3.0),
    (-1.5, -2.5, -4.0),
])
def test_add_various(a, b, expected):
    # Act
    result = add(a, b)
    # Assert
    assert result == expected
```

Рисунок 7. Тест функции сложения

Цель: корректность сложения для целых/вещественных/отрицательных значений.

AAA:

- Arrange — параметры подаются через `parametrize`;
- Act — вызов `add(a, b)`;
- Assert — сравнение с `expected`.

FIRST: тесты независимы, быстры, детерминированы (сложение в Python ассоциативно/коммутативно для этих числовых кейсов).

2) [test_subtraction.py](#) — функция `sub(a, b)`

```
import pytest
from calc.operations import sub

# Arrange
@pytest.mark.parametrize("a,b,expected", [
    (10, 7, 3),
    (3, 8, -5),
    (-2, -5, 3),
    (2.5, 0.5, 2.0),
])
def test_sub_various(a, b, expected):
    # Act
    result = sub(a, b)
    # Assert
    assert result == expected
```

Рисунок 8. Тест функции вычитания

Цель: корректность вычитания и знаков результата.

AAA: Arrange (данные), Act (`sub`), Assert (равенство `expected`).

FIRST: независимость и повторяемость соблюдены.

3) [test_multiplication.py](#) — функция `mul(a, b)`

```
import pytest
from calc.operations import mul

# Arrange
@pytest.mark.parametrize("a,b,expected", [
    (123, 0, 0),
    (0, 999, 0),
    (-4, 2.5, -10.0),
    (-3, -2, 6),
    (1.2, 3, 3.6),
])
def test_mul_various(a, b, expected):
    # Act
    result = mul(a, b)
    # Assert
    assert result == expected
```

Рисунок 9. Тест функции умножения

Цель: корректность умножения, в том числе на ноль и для знаков.

AAA: Arrange (данные), Act (`mul`), Assert (равенство `expected`).

FIRST: быстрые, изолированные проверки.

4) test_division.py — функция `div(a, b)`

```
import pytest
from calc.operations import div

# Базовые случаи деления + десятичные
# Arrange
@pytest.mark.parametrize("a,b,expected", [
    (9, 4, 2.25),
    (-8, 2, -4),
    (7.5, 2.5, 3.0),
])
def test_div_various(a, b, expected):
    # Act
    result = div(a, b)
    # Assert
    assert result == expected

# Деление на ноль — граничный случай с исключением
@pytest.mark.parametrize("a", [0, 1, -5, 3.14])
def test_div_by_zero_raises(a):
    # Arrange / Act / Assert
    with pytest.raises(ZeroDivisionError):
        div(a, 0)
```

Рисунок 10. Тест функции деления

Цель: корректность деления и граничный случай деления на ноль

AAA:

- Arrange — входы через `parametrize`;
- Act — вызов `div`;
- Assert — либо равенство `expected`, либо перехват исключения `with pytest.raises(...)`.

FIRST: чёткая само-валидация, отсутствие побочных эффектов.

5) test_pow_int.py — функция pow_int(base, exp)

```
import pytest
from calc.operations import pow_int

# Целочисленная степень: положительные и отрицательные показатели
# Arrange
@pytest.mark.parametrize("base,exp,expected", [
    (2, 3, 8),
    (2, -1, 0.5),
    (-2, 3, -8),
    (-2, 2, 4),
])
def test_pow_int_various(base, exp, expected):
    # Act
    result = pow_int(base, exp)
    # Assert
    assert result == expected

# Ошибки в pow_int: 0 в отрицательной степени, и тип показателя
@pytest.mark.parametrize("base,exp,exc", [
    (0, -2, ZeroDivisionError),
    (2, 1.5, TypeError),
])
def test_pow_int_errors(base, exp, exc):
    # Arrange / Act / Assert
    with pytest.raises(exc):
        pow_int(base, exp)
```

Рисунок 11. Тест функции степени

Цель: возведение в целую степень с корректной обработкой ошибок.

AAA: Arrange (данные), Act (pow_int), Assert (равенство expected/перехват исключения).

FIRST: предсказуемые и изолированные проверки.

6) [test_factorial.py](#) — функция `factorial(n)`

```
import pytest
from calc.operations import factorial

# Факториал: базовые и малые значения
# Arrange
@pytest.mark.parametrize("n,expected", [
    (0, 1),
    (1, 1),
    (5, 120),
])
def test_factorial_small(n, expected):
    # Act
    result = factorial(n)
    # Assert
    assert result == expected

# Ошибки factorial: отрицательные и неверный тип
# Arrange
@pytest.mark.parametrize("n,exc", [
    (-1, ValueError),
    (2.5, TypeError),
])
def test_factorial_errors(n, exc):
    # Act / Assert
    with pytest.raises(exc):
        factorial(n)
```

Рисунок 12. Тест функции вычисления факториала

Цель: корректность факториала и обработка невалидных входов.

AAA: Arrange (аргументы), Act (`factorial`), Assert (равенство `expected`/перехват исключения).

FIRST: быстрые, детерминированные проверки с чёткими пред- и пост-условиями.

7) [test_properties.py](#) — property-подход для add/mul

```
import pytest
from calc.operations import add, mul

# Коммутативность сложения
@pytest.mark.parametrize("a,b", [(0,0),(1,2),(-3,4),(2.5,-0.5)])
def test_add_commutative(a, b):
    # Arrange / Act
    r1 = add(a, b)
    r2 = add(b, a)
    # Assert
    assert r1 == r2

# Коммутативность умножения
@pytest.mark.parametrize("a,b", [(0,0),(1,2),(-3,4),(2.5,-0.5)])
def test_mul_commutative(a, b):
    # Arrange / Act
    r1 = mul(a, b)
    r2 = mul(b, a)
    # Assert
    assert r1 == r2

# Дистрибутивность  $a*(b+c) == a*b + a*c$ 
@pytest.mark.parametrize("a,b,c", [(1,2,3),(2,-1,5),(0,4,-2),(-2,3,1.5)])
def test_distributive(a, b, c):
    # Arrange / Act
    left = mul(a, add(b, c))
    right = add(mul(a, b), mul(a, c))
    # Assert
    assert left == right
```

Рисунок 12. Property-based testing (PBT)

Цель: показать проверку алгебраических свойств через наборы данных

AAA:

- Arrange/Act — расчёт обеих сторон равенства;
- Assert — сравнение результатов.

FIRST: быстрые свойства без внешних зависимостей, демонстрируют глубину проверок сверх отдельных примеров.

8) test_randomized.py — random-based testing

```
import math
import random
from calc.operations import add, mul

def test_add_mul_randomized_isclose():
    # Arrange
    rng = random.Random(42) # seed

    for _ in range(100):
        a = rng.uniform(-1e3, 1e3)
        b = rng.uniform(-1e3, 1e3)

        # Act
        s = add(a, b)
        p1 = mul(a, 1.0)
        p0 = mul(a, 0.0)

        # Assert
        # (a + b) - a == b с допуском
        assert math.isclose(s - a, b, rel_tol=1e-12, abs_tol=1e-12)
        # a * 1 == a
        assert math.isclose(p1, a, rel_tol=1e-12, abs_tol=1e-12)
        # a * 0 == 0
        assert math.isclose(p0, 0.0, rel_tol=1e-12, abs_tol=1e-12)
```

Рисунок 13. Random-based testing

Цель: дополнительно проверить базовые тождества для `add` и `mul` на случайных данных, сохранив воспроизводимость.

AAA:

- Arrange — фиксируется генератор `rng = Random(42)` и генерируются пары `(a, b)`;
- Act — вычисляются `s = add(a, b)`, `p1 = mul(a, 1)`, `p0 = mul(a, 0)`;

- Assert — проверяются равенства $(a + b) - a \approx b$, $a \cdot 1 \approx a$, $a \cdot 0 \approx 0$ с использованием `math.isclose` (учёт погрешностей float).

FIRST: тест быстрый и изолированный, воспроизводимый за счёт фиксированного seed, самопроверяемый и легко интерпретируемый.

Результаты запуска тестов и метрику code coverage

```
(.venv) PS C:\Users\Lenovo\Desktop\ITMO\3 курс\5 семестр\Тест ПО\Lab1\Software-testing> pytest -c Lab1\pytest.ini Lab1\tests
===== FAILURES =====
test_mul_various[1.2-3-3.6]
a = 1.2, b = 3, expected = 3.6
@pytest.mark.parametrize("a,b,expected", [
    (123, 0, 0),
    (0, 999, 0),
    (-4, 2.5, -10.0),
    (-3, -2, 6),
    (1.2, 3, 3.6),
])
def test_mul_various(a, b, expected):
    # Act
    result = mul(a, b)
    # Assert
    assert result == expected
E       assert 3.5999999999999996 == 3.6
Lab1\tests\test_multiplication.py:16: AssertionError
===== tests coverage =====
coverage: platform win32, python 3.13.2-final-0
Name                               Stmts Miss Cover Missing
-----
Lab1\calc\__init__.py               0      0 100%
Lab1\calc\operations.py             27      0 100%
TOTAL                               27      0 100%
===== short test summary info =====
FAILED Lab1\tests\test_multiplication.py::test_mul_various[1.2-3-3.6] - assert 3.5999999999999996 == 3.6
1 failed, 43 passed in 0.13s
```

Рисунок 14. Результаты запуска тестов и code coverage

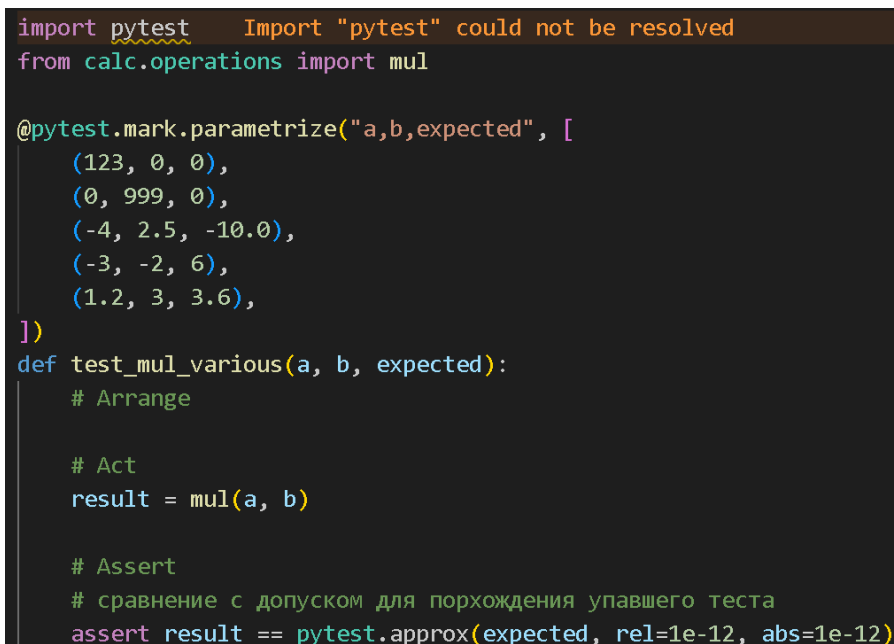
Как видно на рисунке 14, из 44 тестов успешно прошли 43, и один тест завершился падением.

Причина падения не в логике функции `mul(a, b)`, а в самой проверке: сравнение выполнялось на строгие равенства для чисел с плавающей точкой. В результате проявилась стандартная особенность двоичного представления десятичных дробей: произведение $1.2 * 3$

возвращает значение 3.5999999999999996, которое математически эквивалентно 3.6, но побитово не совпадает с ним.

При этом показатель покрытия кода составил 100% для модуля `calc/operations.py`, что означает, что все строки и ветви были выполнены тестами, а единственная проблема относится к методике утверждения результата в тесте, а не к реализации операций.

Коррекция упавшего теста



```
import pytest    Import "pytest" could not be resolved
from calc.operations import mul

@pytest.mark.parametrize("a,b,expected", [
    (123, 0, 0),
    (0, 999, 0),
    (-4, 2.5, -10.0),
    (-3, -2, 6),
    (1.2, 3, 3.6),
])
def test_mul_various(a, b, expected):
    # Arrange

    # Act
    result = mul(a, b)

    # Assert
    # сравнение с допуском для прохождения упавшего теста
    assert result == pytest.approx(expected, rel=1e-12, abs=1e-12)
```

Рисунок 15. Тест функции умножения с допуском

Как показано на рисунке 15, для прохождения последнего теста была изменена стратегия сравнения результатов умножения с ожидаемыми значениями: вместо строгого равенства используется сравнение с допуском (через `pytest.approx`), что является корректной практикой для проверок с участием `float`.

```
(.venv) PS C:\Users\Lenovo\Desktop\ITMO\3 курс\5 семестр\Тест ПО\Lab1\Software-testing> pytest -c Lab1\pytest.ini Lab1\tests
===== tests coverage ===== [100%]
coverage: platform win32, python 3.13.2-final-0
Name                               Stmts  Miss  Cover   Missing
-----
Lab1\calc\__init__.py               0      0  100%
Lab1\calc\operations.py            27      0  100%
TOTAL                               27      0  100%
44 passed in 0.10s
```

Рисунок 16. Результаты запуска тестов и code coverage после правки

После такой правки повторный прогон набора приводит к полностью зелёному результату (44 из 44 тестов проходят), при этом показатель покрытия остаётся на уровне 100%, что подтверждает достаточность тестового набора для дальнейших шагов, включая проведение мутационного тестирования.

Mutation testing

MutPy mutation report

29.09.2025 21:12

Target

- `calc`

Tests [44]

- `test_division` [0.08 s]
- `test_pow_int` [0.06 s]
- `test_properties` [0.065 s]
- `conftest` [0.052 s]
- `test_subtraction` [0.057 s]
- `test_addition` [0.058 s]
- `test_factorial` [0.059 s]
- `test_multiplication` [0.059 s]

Result summary

- **Score** - 90.9%
- **Time** - 4.1 s

Mutants [26]

- **killed** - 20
- **survived** - 2
- **incompetent** - 4
- **timeout** - 0

Рисунок 17. Результаты mutation testing

Мутационное тестирование выполнялось с помощью библиотеки MutPy, интегрированной в [GitHub Actions](#). В пайплайне используется Python 3.10; перед запуском мутантов прогоняются все юнит-тесты (clean trial).

MutPy модифицирует исходники на уровне AST, применяя операторные мутации (например, ROR — замены сравнений, AOR — замены арифметических операторов и др.), после чего повторно запускает тесты для каждого мутанта. Если тесты падают — мутант «убит», если тесты остаются зелёными — мутант «выжил». Результаты выгружаются как artifacts: [текстовый отчёт](#) и [HTML-отчёт](#), в котором легче рассматривать результаты.

Итог прогона:

- 44 юнит-теста успешно проходят (clean trial), покрытие базовыми тестами: 100%.
- Сгенерировано 26 мутантов: 20 «убито», 2 «выжили», 4 «incompetent» (некорректные мутации).
- Mutation score $\approx 90,9$ %.

Локализация выживших мутантов: защита в функции `pow_int` для нулевого основания (ветка “`base == 0` и `exp < 0`”). Тип мутации — ROR (замена оператора сравнения), что подсветило недостающие граничные сценарии: (0, 0), (0, положительный `exp`) и (0, отрицательный `exp` приведет к исключению). Эти случаи добавлены отдельными тестами в следующем подразделе.

Добивание выживших мутантов ([pow_int](#))

```
def test_pow_int_zero_base_zero_exp():
    # Arrange
    base, exp = 0, 0
    # Act + Assert
    assert pow_int(base, exp) == 1 # в Python 0**0 == 1

def test_pow_int_zero_base_positive_exp():
    # Arrange
    base, exp = 0, 5
    # Act + Assert
    assert pow_int(base, exp) == 0

def test_pow_int_zero_base_negative_exp_raises():
    # Arrange / Act + Assert
    with pytest.raises(ZeroDivisionError):
        pow_int(0, -1)
```

Рисунок 18. Дополнительные проверки в тесте функции степени

Как видно на рисунке 18, чтобы “докусать” выживших, я добавил три точечных теста:

- $0^{**}0$ дает 1 — фиксирует семантику Python для нулевой степени;
- $0^{**}k$, $k > 0$ дает 0 — корректное значение при положительном показателе;
- $0^{**}(-1)$ приводит к выбросу `ZeroDivisionError` (запрет нулевого основания в отрицательной степени).

Тесты оформлены по AAA, быстрые и изолированные (FIRST), поэтому легко воспроизводимы и не создают шум.

MutPy mutation report

🕒 29.09.2025 22:24

Target

- `calc`

Tests [48]

- `test_division` [0.08 s]
- `test_pow_int` [0.064 s]
- `test_properties` [0.074 s]
- `conftest` [0.054 s]
- `test_subtraction` [0.059 s]
- `test_randomized` [0.057 s]
- `test_addition` [0.062 s]
- `test_factorial` [0.062 s]
- `test_multiplication` [0.062 s]

Result summary

- 📊 **Score** - 100.0%
- 🕒 **Time** - 4.4 s

Mutants [26]

- `killed` - 22
- `survived` - 0
- `incompetent` - 4
- `timeout` - 0

Рисунок 19. Результаты mutation testing после поправки

Как видно на рисунке 19, после добавления этих проверок [повторный запуск](#) MutPy дал: 26 мутантов, из них 22 killed, 0 survived, 4 incompetent, timeout 0; mutation score = 100%, общее время ≈ 4.4 s.

Таким образом, набор тестов теперь покрывает не только “счастливые” и граничные сценарии, но и устойчив к мелким дефектам в условиях и сравнительных операторах.

ЗАКЛЮЧЕНИЕ

В ходе работы был разработан и воспроизводимо запущен полный цикл модульного тестирования мини-проекта [«calc»](#).

Построен набор юнит-тестов на `pytest` с явной схемой AAA и соблюдением принципов FIRST: быстрые, изолированные, повторяемые и самопроверяемые проверки без внешней инфраструктуры. Для сокращения дублирования входов применён data-driven подход (`parametrize`), добавлены property-тесты для алгебраических свойств и randomized-тест с фиксированным `seed`.

Метрики качества:

- code coverage по модулю [calc/operations.py](#): 100%;
- mutation testing (MutPy): mutation score 100% (все релевантные мутанты «убиты», 4 — «incompetent»).

В процессе работы были выявлены и исправлены тонкости сравнения чисел с плавающей точкой (замена строгого равенства на сравнение с допуском), а также добавлены точечные проверки для краевых случаев функции `pow_int` (нулевое основание и различные показатели), что окончательно “закрыло” выживших мутантов.

Таким образом, текущий набор тестов покрывает как корректные, так и граничные сценарии и демонстрирует устойчивость к типичным дефектам на уровне операторов и условий.