# Написание Тестов на Python

# pytest, unittest, hypothesis

# Введение: Зачем Тестировать?

### Раннее Обнаружение Ошибок

Выявление дефектов на самых ранних этапах разработки.

### Гарантия Работоспособности (Регрессия)

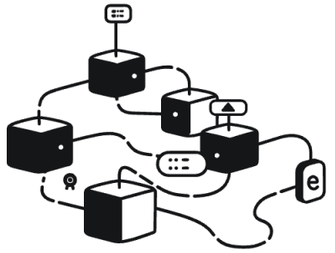
Уверенность в том, что изменения не сломали существующую функциональность.

### Тесты как Документация

Тесты показывают, как код должен работать в различных сценариях.

### Чистый и Модульный Код

Стимулирование написания легко тестируемых, изолированных компонентов.



### Пирамида Тестирования

- Unit-тесты: Быстрые, проверяют отдельные функции/классы (наш фокус).
- Системные (E2E): Проверяют систему целик

# unittest — Стандартный Подход

**Встроенный модуль, следующий парадигме xUnit (унаследован от JUnit).**

### 1.Классовая Структура

Тесты – это методы класса, наследующего от *`unittest.TestCase()`*

### 2.Специальные Assert-Методы

Использование ***self.assertEqual(),*** ***self.assertTrue()*** и.т.д

### 3.Фикстуры (setUp/tearDown)

Методы для подготовки и очистки данных до/после каждого теста или класса.

### Плюсы

- Стандартная библиотека Python.
- Строгая, понятная структура.

### Минусы

- Много boilerplate-кода (наследование, *`self`*).
- Менее читаемый синтаксис по сравнению с pytest.

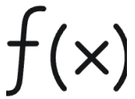
# Пример Кода на unittest

```python
import unittest

def add(a, b):  3 usages
    return a + b

class TestMathOperations(unittest.TestCase):
    def setUp(self):
        self.num1 = 5
        self.num2 = 3

    def test_add_integers(self):
        self.assertEqual(add(self.num1, self.num2),  second: 8)

    def test_add_negative(self):
        self.assertEqual(add(-1, -1), -2)

    def test_add_wrong_type(self):
        with self.assertRaises(TypeError):
            add( a: "5",  b: 3)

if __name__ == '__main__':
    unittest.main()
```

```
Ran 3 tests in 0.062s

OK
```

# pytest — Популярный и Мощный Выбор

Сторонний фреймворк, ставший стандартом де-факто благодаря своей гибкости и лаконичности.



### Простые Функции

Нет необходимости в классах. Тест – любая функция, начинающаяся с *`test_`*.

### Мощные Фикстуры

Гибкая система фикстур (*`@pytest.fixture`*) для переиспользуемой подготовки данных.

### Стандартные Assert

Используйте обычные выражения *`assert`*. Не нужно запоминать специальные методы.

### Параметризация

Легкий запуск одного теста с разными наборами данных *(`@pytest.mark.parametrize`).*

# Пример Кода на pytest

Лаконичность, читаемость и использование мощных фич pytest.

```python
import pytest

def add(a, b):  3 usages
    return a + b

@pytest.fixture  2 usages
def sample_data():
    return (5, 3, 8)


def test_add_with_fixture(sample_data):
    a, b, expected = sample_data
    assert add(a, b) == expected

@pytest.mark.parametrize("a, b, expected", [
    (5, 3, 8),
    (-1, -1, -2),
    (0, 0, 0)
])
def test_add_parametrized(a, b, expected):
    assert add(a, b) == expected

def test_add_type_error():
    with pytest.raises(TypeError):
        add( a: "5",  b: 3)

if __name__ == "__main__":
    pytest.main()
```

**Вывод:**

```
============================= test session starts ==============================
collecting ... collected 5 items

BB.py::test_add_with_fixture PASSED                                      [ 20%]
BB.py::test_add_parametrized[5-3-8] PASSED                               [ 40%]
BB.py::test_add_parametrized[-1--1--2] PASSED                            [ 60%]
BB.py::test_add_parametrized[0-0-0] PASSED                               [ 80%]
BB.py::test_add_type_error PASSED                                        [100%]

============================== 5 passed in 0.02s ===============================

Process finished with exit code 0
```

# Часть 3: hypothesis — Тестирование на Стероидах

## Переход от Example-Based к Property-Based Тестированию

### Проблема Example-Based

Мы придумываем примеры вручную. Легко упустить краевые случаи (очень большие числа, `None`, Unicode, пустые строки).

### Решение: Property-Based

Мы описываем **свойство (инвариант)** функции, которое должно выполняться всегда.

### Автоматическая Генерация

Hypothesis автоматически генерирует сотни случайных данных, чтобы найти нарушение этого свойства.

### Поиск Минимального Примера

Если ошибка найдена, Hypothesis показывает минимальный набор данных, приводящий к сбою (шаг воспроизведения).

# hypothesis в Действии

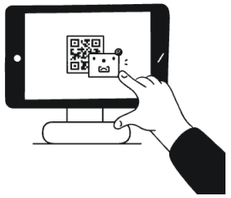
Использование стратегий для генерации данных и проверки фундаментальных свойств кода.

```python
from hypothesis import given, strategies as st

def add(a, b):  2 usages
    return a + b


def encode(string):  1 usage
    return string[::-1]

def decode(string):  1 usage
    return string[::-1]

@given(st.text())
def test_encode_decode_inversion(s):
    assert decode(encode(s)) == s

@given(st.integers(), st.integers())
def test_add_commutative(a, b):
    assert add(a, b) == add(b, a)
```

**Вывод:**

```
============================= test session starts ==============================
collecting ... collected 2 items

EE.py::test_encode_decode_inversion PASSED                               [ 50%]
EE.py::test_add_commutative PASSED                                       [100%]

============================== 2 passed in 0.55s ===============================

Process finished with exit code 0
```

# Заключение



Оптимальный путь к надежному коду.

### Начните с pytest
Самый удобный и распространенный инструмент для большинства задач.

### Изолированность
Каждый тест должен проверять одну вещь. Используйте фикстуры для подготовки данных.

### Параметризация
Применяйте параметризацию для тестирования одинаковой логики на разных входных данных.

### Краевые Случаи
Не забывайте тестировать исключения и граничные условия (ноль, пустые значения, лимиты).

### Подключите hypothesis
Используйте property-based тестирование для критически важных функций, где важны фундаментальные св