**Установка pytest**

bash

# Установка фреймворка pytest для тестирования

pip install pytest

# Проверка успешной установки и версии

pytest --version

# Альтернативный способ запуска (если команда pytest не работает)

python -m pytest --version

**📁 Структура проекта**

text

my\_project/ # Корневая папка проекта

├── src/ # Папка для исходного кода приложения

│ ├── \_\_init\_\_.py # Файл, превращающий папку в Python-пакет

│ └── calculator.py # Модуль с основной логикой приложения

├── tests/ # Папка для всех тестов

│ ├── \_\_init\_\_.py # Делает tests Python-пакетом

│ └── test\_calculator.py # Тесты для calculator.py

├── requirements.txt # Файл со списком зависимостей проекта

└── pytest.ini # Файл конфигурации для pytest

**✍️ Создание первого теста**

**Модуль с кодом приложения (src/calculator.py)**

python

def add(a: float, b: float) -> float:

"""Сложение двух чисел"""

return a + b # Просто возвращаем сумму аргументов

def multiply(a: float, b: float) -> float:

"""Умножение двух чисел"""

return a \* b # Возвращаем произведение аргументов

def divide(a: float, b: float) -> float:

"""Деление двух чисел"""

if b == 0: # Проверяем, не является ли делитель нулем

# Если делитель ноль, выбрасываем исключение

raise ValueError("Деление на ноль невозможно")

return a / b # Возвращаем результат деления

**Тестовый модуль (tests/test\_calculator.py)**

python

# Импортируем pytest для использования специальных функций

import pytest

# Импортируем функции, которые будем тестировать

from src.calculator import add, multiply, divide

# Создаем класс для группировки связанных тестов

class TestCalculator:

"""Тесты для калькулятора"""

def test\_add\_positive\_numbers(self):

"""Тест сложения положительных чисел"""

# Проверяем, что 2 + 3 = 5

assert add(2, 3) == 5

# Проверяем, что 0 + 0 = 0

assert add(0, 0) == 0

def test\_add\_negative\_numbers(self):

"""Тест сложения отрицательных чисел"""

# Проверяем сложение двух отрицательных чисел

assert add(-1, -1) == -2

# Проверяем сложение отрицательного и положительного

assert add(-5, 3) == -2

def test\_multiply\_basic(self):

"""Тест умножения"""

# Проверяем умножение положительных чисел

assert multiply(3, 4) == 12

# Проверяем умножение на ноль

assert multiply(0, 5) == 0

# Проверяем умножение отрицательного на положительное

assert multiply(-2, 3) == -6

def test\_divide\_normal(self):

"""Тест обычного деления"""

# Проверяем деление без остатка

assert divide(10, 2) == 5

# Проверяем деление с дробным результатом

assert divide(5, 2) == 2.5

def test\_divide\_by\_zero(self):

"""Тест деления на ноль"""

# Используем менеджер контекста для проверки исключений

with pytest.raises(ValueError, match="Деление на ноль невозможно"):

# Этот код должен вызвать ValueError

divide(10, 0)

**🚀 Запуск тестов**

**Основные команды запуска**

bash

# Запуск всех тестов в проекте

pytest

# Подробный вывод (verbose) - показывает названия всех тестов

pytest -v

# Запуск тестов только из конкретного файла

pytest tests/test\_calculator.py

# Запуск тестов только из конкретного класса

pytest tests/test\_calculator.py::TestCalculator

# Запуск конкретного тестового метода

pytest tests/test\_calculator.py::TestCalculator::test\_add\_positive\_numbers

# Запуск тестов, содержащих "add" в названии

pytest -k "add"

# Показать вывод print() функций в тестах

pytest -s

# Остановиться при первой ошибке (x - сокращение от exit)

pytest -x

**Полезные флаги для расширенной работы**

bash

# Измерение покрытия кода тестами (требуется pytest-cov)

pytest --cov=src

# Генерация HTML отчета о тестировании (требуется pytest-html)

pytest --html=report.html

# Параллельный запуск тестов в 4 потока (требуется pytest-xdist)

pytest -n 4

# Показать 5 самых медленных тестов

pytest --durations=5

**🛠 Основные возможности pytest**

**1. Умные утверждения (assert)**

python

# pytest автоматически покажет значения при ошибке

assert result == expected # Сравнение на равенство

assert user.is\_active # Проверка булевого значения

assert len(items) > 0 # Проверка длины коллекции

**2. Автоматическое обнаружение тестов**

* **Файлы**: test\_\*.py или \*\_test.py (pytest ищет файлы по этим шаблонам)
* **Функции**: test\_\* (любая функция, начинающаяся с test\_)
* **Классы**: Test\* (любой класс, начинающийся с Test)

**3. Подробные отчеты об ошибках**

При падении теста pytest показывает:

* Значения переменных в момент ошибки
* Разницу между ожидаемым и фактическим результатом
* Полный traceback ошибки

**🔧 Фикстуры (Fixtures)**

**Простая фикстура для подготовки данных**

python

import pytest

# Декоратор @pytest.fixture отмечает функцию как фикстуру

@pytest.fixture

def sample\_user():

"""Фикстура создает тестового пользователя"""

# Возвращаем готовые данные для тестов

return {

"name": "John Doe",

"email": "john@example.com",

"age": 30

}

# Фикстура sample\_user автоматически передается в тест

def test\_user\_age(sample\_user):

"""Тест использует фикстуру sample\_user"""

# Проверяем возраст пользователя

assert sample\_user["age"] == 30

# Проверяем имя пользователя

assert sample\_user["name"] == "John Doe"

**Фикстура с setup и teardown**

python

@pytest.fixture

def database\_connection():

"""Фикстура с подготовкой и очисткой ресурсов"""

# Setup - код выполняется ДО теста

connection = connect\_to\_database() # Создаем подключение

print("Подключение к БД установлено")

yield connection # Передаем подключение в тест

# Teardown - код выполняется ПОСЛЕ теста

connection.close() # Закрываем подключение

print("Подключение к БД закрыто")

def test\_database\_query(database\_connection):

"""Тест использует фикстуру с БД"""

# Выполняем запрос к базе данных

result = database\_connection.execute("SELECT 1")

# Проверяем, что результат не None

assert result is not None

**Автоматически используемая фикстура**

python

@pytest.fixture(autouse=True) # autouse=True - фикстура запускается автоматически

def setup\_environment():

"""Запускается автоматически для каждого теста"""

print("Начало теста") # Выполняется перед каждым тестом

yield # Разделитель между setup и teardown

print("Конец теста") # Выполняется после каждого теста

**📊 Параметризация тестов**

**Параметризация одного теста разными данными**

python

import pytest

# Декоратор parametrize позволяет запустить один тест с разными данными

@pytest.mark.parametrize("a,b,expected", [

(1, 2, 3), # Первый набор данных: a=1, b=2, expected=3

(0, 0, 0), # Второй набор данных

(-1, 1, 0), # Третий набор данных

(10, -5, 5) # Четвертый набор данных

])

def test\_add\_parametric(a, b, expected):

"""Один тест с разными наборами данных"""

from src.calculator import add

# Тест запустится 4 раза с разными a, b, expected

assert add(a, b) == expected

**Параметризация с понятными идентификаторами**

python

@pytest.mark.parametrize("a,b,expected", [

(1, 2, 3), # Будет показано как "positive" в отчете

(0, 0, 0), # Будет показано как "zeros" в отчете

(-1, 1, 0) # Будет показано как "negative\_positive" в отчете

], ids=["positive", "zeros", "negative\_positive"]) # Понятные названия тестов

def test\_add\_with\_ids(a, b, expected):

"""Тест с читаемыми названиями для каждого набора данных"""

assert add(a, b) == expected

**⚙️ Настройка конфигурации**

**Файл pytest.ini**

ini

[pytest]

# Настройка обнаружения тестовых файлов

python\_files = test\_\*.py # Искать файлы, начинающиеся с test\_

python\_classes = Test\* # Искать классы, начинающиеся с Test

python\_functions = test\_\* # Искать функции, начинающиеся с test\_

# Опции по умолчанию для всех запусков

addopts = -v --tb=short # -v: подробный вывод, --tb=short: короткие traceback

# Папки, где искать тесты

testpaths = tests

# Определение пользовательских маркеров

markers =

slow: медленные тесты # Маркер для медленных тестов

integration: интеграционные тесты # Маркер для интеграционных тестов

**Файл pyproject.toml (альтернатива)**

toml

[tool.pytest.ini\_options]

# Настройки обнаружения тестов

python\_files = ["test\_\*.py"]

python\_classes = ["Test\*"]

python\_functions = ["test\_\*"]

# Опции по умолчанию

addopts = ["-v", "--tb=short"]

# Папки с тестами

testpaths = ["tests"]

# Пользовательские маркеры

[tool.pytest.ini\_options.markers]

slow = "медленные тесты"

integration = "интеграционные тесты"

**🏷 Маркеры (Markers)**

**Создание и использование маркеров**

python

import time

# Помечаем тест как медленный с помощью декоратора @pytest.mark.slow

@pytest.mark.slow

def test\_long\_operation():

"""Этот тест будет помечен как медленный"""

time.sleep(5) # Имитируем долгую операцию

assert True # Проверка

# Помечаем тест как интеграционный

@pytest.mark.integration

def test\_database\_integration():

"""Интеграционный тест с внешними системами"""

assert True # Здесь была бы реальная интеграция с БД

**Запуск тестов по маркерам**

bash

# Запустить все тесты, НЕ помеченные как "slow"

pytest -m "not slow"

# Запустить только тесты, помеченные как "integration"

pytest -m integration

# Запустить только тесты, помеченные как "slow"

pytest -m slow

**🐛 Частые проблемы и решения**

**Проблема 1: Импорт модулей**

**Ошибка**: ModuleNotFoundError: No module named 'src'

**Решение**:

bash

# Установите переменную PYTHONPATH на текущую директорию

export PYTHONPATH=.

# Или используйте альтернативный способ запуска

python -m pytest

**Проблема 2: Кодировка файлов**

**Ошибка**: SyntaxError: source code string cannot contain null bytes

**Решение**:

python

# Создавайте файлы с явным указанием кодировки UTF-8

with open('file.py', 'w', encoding='utf-8') as f:

f.write('код') # Записываем код в файл

**Проблема 3: Тесты не находятся**

**Решение**:

* Убедитесь, что файлы начинаются с test\_
* Убедитесь, что функции начинаются с test\_
* Проверьте правильность структуры проекта

**📈 Best Practices (Лучшие практики)**

**1. Правила именования**

python

# Тестовые файлы: test\_<module>.py

test\_calculator.py

test\_user\_service.py

# Тестовые функции: test\_<scenario>

test\_add\_numbers()

test\_user\_creation()

# Тестовые классы: Test<Feature>

TestCalculator

TestUserAuthentication

**2. Организация тестов по шаблону**

python

def test\_<что\_тестируем>\_<условия>\_<ожидаемый\_результат>():

# Примеры:

test\_add\_positive\_numbers\_returns\_sum()

test\_user\_login\_with\_invalid\_password\_raises\_error()

test\_database\_connection\_with\_valid\_config\_succeeds()

**3. Независимость тестов**

python

# Каждый тест должен быть самостоятельным

def test\_user\_creation():

user = create\_user() # Создаем нового пользователя

assert user.id is not None

def test\_user\_deletion():

user = create\_user() # Снова создаем нового пользователя

delete\_user(user.id) # Не зависим от предыдущего теста

assert user\_not\_exists(user.id)

**4. Читаемость тестов**

python

# ХОРОШО: понятные названия и структура

def test\_user\_creation\_with\_valid\_data():

# Создаем пользователя с явными параметрами

user = create\_user(name="John", email="john@test.com")

# Проверяем конкретные утверждения

assert user.id is not None # ID должен быть установлен

assert user.is\_active # Пользователь должен быть активен

# ПЛОХО: непонятные названия и сжатый код

def test\_user():

u = create\_user("John", "john@test.com") # Неясные параметры

assert u.id and u.active # Неявные проверки

**🎓 Пример полного проекта**

**Структура проекта**

text

project/ # Корневая директория проекта

├── src/ # Исходный код приложения

│ ├── \_\_init\_\_.py # Делает src Python-пакетом

│ ├── calculator.py # Модуль с математическими функциями

│ └── user.py # Модуль для работы с пользователями

├── tests/ # Все тесты проекта

│ ├── \_\_init\_\_.py # Делает tests Python-пакетом

│ ├── test\_calculator.py # Тесты для calculator.py

│ ├── test\_user.py # Тесты для user.py

│ └── conftest.py # Общие фикстуры для всех тестов

├── pytest.ini # Конфигурация pytest

└── requirements.txt # Список зависимостей проекта

**Файл с общими фикстурами (tests/conftest.py)**

python

import pytest

# Фикстуры в conftest.py доступны во всех тестах проекта

@pytest.fixture

def sample\_user\_data():

"""Фикстура возвращает тестовые данные пользователя"""

return {

"name": "Test User", # Тестовое имя

"email": "test@example.com", # Тестовый email

"password": "secure123" # Тестовый пароль

}

# Эта фикстура может использоваться в test\_user.py, test\_calculator.py и др.