Шпаргалка: pytest

Что такое pytest

Pytest — это фреймфорк для автотестов на Python.

Фреймворк похож на каркас для тестов. В нём уже есть готовые решения, которые можно использовать. Без него пришлось бы писать больше одинакового кода.

Пример. Автотест проверяет, что 2 + 2 = 4.

👎 Вот автотест без pytest:

```
def test_example():

    sum = 2 + 2
    if sum != 4:
       raise AssertionError('False!')

    else:
        print('Test is passed')

test_example()
```

👍 То же самое, но с pytest:

```
def test_example():
    sum = 2 + 2
    assert sum == 4
```

Структура теста в pytest

По сути юнит-тест — это функция, которая сравнивает ожидаемый результат и фактический. Её нужно объявить, написать тело, а потом вызвать.

📢 **Структура теста:**

1. Объявить метод с ключевым словом test.
2. Написать тело метода: создать объект класса, вызвать метод, создать и передать туда аргументы.
3. Сравнить фактический результат и ожидаемый.

Пример. Нужно написать юнит-тест для метода `is_adult` класса `Person`.

Метод `is_adult` получает на вход возраст человека `age`. Он возвращает `True`, если возраст больше или равен 18. Иначе — `False`.

Вот как выглядит тест для метода:

```
def test_is_adult_when_age_is_more_than18_true():

    person = Person()
    age = 20
    actual_result = person.is_adult(age)
    # проверка результата actual_result с помощью assertTrue;
```

```
    # если actual_result — true, тест пройден успешно
    assert actual_result
```

Что такое assert

Любой тест проверяет, как работает система. Для этого нужно сравнить два значения: ожидаемый результат и фактический.

В pytest для этого есть команда `assert`. Она работает как логическое выражение: возвращает `True` или `False`.

Как использовать `assert`

Представь, что проверяешь калькулятор: нужно убедиться, что он правильно складывает однозначные целые числа.

Сначала нужно создать тест-кейс с шагами воспроизведения.

✅ **Калькулятор правильно складывает однозначные целые числа**

Шаги:

1. Ввести 5.
2. Нажать `+`.
3. Ввести 6.
4. Нажать `=`.

ОР: Результат сложения — 11.

Чтобы перевести такой тест-кейс в программу, нужно последовательно проходить по шагам и переводить их в код.

В конце нужно проверить, что результат действительно равен 11. Для этого можно включить в автотест выражение `assert`. Ещё понадобится оператор сравнения `==`. Получится вот так:

```
assert result == 11
```

Если утверждение верно, тест пройден. Если утверждение неверно, произойдёт ошибка.

Ошибка `AssertionError`

Если проверка не прошла, программа вернёт сообщение об ошибке — `AssertionError`.

Тесты с `AssertionError` получают статус `Failed`. Если в коде есть другие проверки, программа остановится здесь.

В ошибке pytest выводит ожидаемый и фактический результаты. Вот так:

```
 assert 5 == 4
    +5
    -4

test__calc.py:40: AssertionError
```

Базовые assert

С `assert` можно использовать операторы `=`, `!=`, `>`, `<`, `=>`, `<=`, `or` и `and`. Они помогают писать разные проверки.

Проверка равенства: `==`

Помогает проверить, что одно значение равно другому.

```
assert result == 11
```

Со строками тоже работает.

Проверка неравенства: `!=`

Помогает убедиться, что одно значение не равно другому. Например, результат сложения не равен 6:

```
assert result != 6.
```

Чаще всего эту проверку используют, когда есть большой массив данных. Проще проверить, что результат не равен неправильному, чем проверять на равенство всем правильным вариантам.

Проверка «больше» и «меньше»: `>` и `<`

Например, нужно проверить, что баланс больше нуля. Понадобится оператор `>`:

```
def test_deposit_more_than_zero():

    # получение данных из базы и сохранение их в переменную
    deposit = get_user_deposit()
    # проверка, что баланс пользователя больше 0
    assert deposit > 0
```

Проверки с `=>` и `<=` работают точно так же.

Проверки с `or` и `and`

Операторы `or` и `and` помогают проверить несколько условий сразу. Например, данные не равны `None` и пустому списку: `len(data) != 0 and data != None`.

```
def test_correct_data_list_from_db_true():

    # получение данных из базы и сохранение их в переменную
    data = get_user_data()
    # проверка, что данные есть и длина списка с данными не равна 0
    assert len(data) != 0 and data != None
```

Проверить, есть ли элемент в списке: `in`

Чтобы проверить, есть ли в списке определённое значение, используй оператор `in`.

Функция `get_user_role()` возвращает текущую роль пользователя. Так можно проверить, что она есть в писке допустимых `correct_roles`.

```
def test_correct_user_role_true():

    correct_roles = ['guest', 'user', 'admin', 'moderator']
    # получение данных из базы и сохранение их в переменную
    user_role = get_user_role()
    # проверка, что в массиве допустимых ролей содержится полученная от сервера роль
    assert user_role in correct_roles
```

Как покрыть тестами метод

Чтобы полностью покрыть метод, нужно вызвать метод с разными аргументами и проверить, корректно ли он работает в разных ситуациях.

Пример. Метод `count_list_elements(arg)` подсчитывает, сколько элементов в списке, и возвращает это число. Ещё он проверяет, что в качестве аргумента передают именно список:

```
def count_list_elements(arg):    # объявление метода с принимаемым аргументом в виде списка

    if type(arg) is not list:      # проверка, какой тип аргумента
        return 'Not list!'

    elif arg == []:           # проверка, не пустой ли это список (в таком случае можно не считать элементы)
        return 'List is empty!'

    else:                     # если аргумент — список и он не пустой, можно подсчитать количество его элементов
        count = 0             # объявление переменной, которая будет считать количество элементов в списке
        for i in arg:              # перебор циклом всех элементов списка
            count = count+1        # увеличение переменной count на единицу за каждый элемент списка
        return count               # возвращение из метода количества элементов списка
```

Чтобы покрыть этот метод тестами, его нужно вызвать с такими аргументами:

- в метод передаётся список с несколькими элементами;
- в метод передаётся не список — например, строка. Тогда метод должен вывести `Not list!`;
- в метод передаётся пустой список. Должно появиться `List is empty`.

Сценарий №1. Позитивный тест

Проще всего начинать с позитивной проверки — такого теста, который выполняет метод корректно. Например, в метод `count_list_elements(arg)` передаётся список с тремя элементами:

```
def test_count_list_elements_correct_list():  # называем тест

    arg = [1,2,3]                  # объявление переменной, содержащей список, который будет передаваться как аргумент в тестируемый метод
    assert count_list_elements(arg) == 3      # проверка, что количество элементов в списке равно трём
```

Сценарий №2. Негативный тест

После позитивного теста можно выполнить простой негативный. Например, передать в `count_list_elements(arg)` не список, а любой другой тип данных — строку, число, словарь, `None`.

```
def test_count_list_elements_not_list():  # называем тест

    not_list = 1                # объявление переменной, тип которой - не список
    assert count_list_elements(not_list) == 'Not list!' # проверка, что метод возвращает корректный ответ, если в него придёт не список
```

Сценарий №3. Пустой список

В качестве аргумента передаётся пустой список. Тогда можно не запускать код, который будет считать элементы — считать нечего:

```
def test_count_list_elements_empty_list(): # называем тест

    empty_list = [] # объявили переменую в виде пустого списка
    # проверка, что метод возвращает корректный ответ в случае передачи
    # в него пустого списка
    assert count_list_elements(empty_list ) == 'List is empty!'
```

Тестовое покрытие кода

Тестовое покрытие кода показывает, какой процент кода покрыт автотестами. Рассчитывают его так:

📢 **TestCoverage = (TestCode/TotalCode) * 100%**

Здесь:

- **TestCoverage** — высчитываемое тестовое покрытие кода;
- **TestCode** — строки кода, которые покрыты тестами;
- **TotalCode** — общее количество строк кода.

Пример. Нужно рассчитать процент тестового покрытия кода метода `count_list_elements`. Всего в методе 10 строк кода.

```
def count_list_elements(arg):   # 1

    if type(arg) is not list:   # 2
        return 'Not list!'      # 3

    elif arg == []:             # 4
        return 'List is empty!' # 5

    else:                       # 6
        count = 0               # 7
        for i in arg:           # 8
            count = count+1     # 9
        return count            # 10
```

Посчитаем, сколько строк кода покрывает первый позитивный тест `test_count_list_elements_correct_list`. В этом тесте вызываются строки кода:

- 1 — объявление метода;
- 2 — проверка на тип аргумента;
- 4 — проверка на пустой список;
- 6-10 — блок кода, подсчитывающий количество элементов.

Получается, что не покрытыми остаются 3 и 5 строки кода. Их покрывают остальные тесты: `test_count_list_elements_not_list` — третью, `test_count_list_elements_empty_list` — пятую.

Три теста затрагивают 100% кода метода.

Как покрыть тестами класс

📢 Протестировать класс — это покрыть тестами всё, что внутри него. Все методы, параметры и переменные.

Пример. Есть класс `Dog`, который создаёт собачек. Он содержит метод, который увеличивает уровень счастья собачки — `pet_the_dog(self)`.

```
class Dog:

    def __init__(self, name):  # констуктор класса, который в качестве аргумента принимает имя собачки
        self.name = name
        self.happiness = 50    # уровень счастья у новой собачки всегда 50

    def pet_the_dog(self):     # метод, который увеличивает уровень счастья собачки на 10%
        self.happiness += 10
```

1. Создать тестовый класс

Сперва нужно создать тестовый класс в отдельном файле:

```
class TestDog:
```

Чтобы создать тестовый класс, его нужно назвать.

Как назвать тестовый класс в pytest:

Есть три правила:

- Классы должны начинаться с ключевого слова `Test`. Так pytest поймёт, что они тестовые.
- Название отражает суть. Можно написать просто имя класса. Например, `Dog` или `Registration`.
- Написано в стиле CamelCase. Например, `TestRegistration` или `TestLogin`.

2. Протестировать метод `__init__`

Дальше нужно протестировать все методы внутри класса. А для этого создать объект, иначе обратиться к методам не получится.

В первую очередь нужно протестировать метод `__init__`: он выполнится первым, когда в классе появится объект.

Для этого нужно проверить каждое из полей. В примере с собаками это имя и уровень счастья. Вот тест, который проверяет, правильно ли собачке присвоили имя:

```
class TestDog:

    def test_name_of_dog_true(self):

        dog = Dog('Sharik')   # создание нового экземпляра класса

        assert dog.name == 'Sharik'  # проверка корректности имени в созданном экземпляре
```

Имя протестировали.

Дальше больше: у каждого экземпляра класса есть ещё один параметр: уровень счастья. По умолчанию его значение — 50. Это тоже нужно проверить:

```
def test_default_value_happiness_true(self):

        # создание нового экземпляра класса
        dog = Dog('Sharik')

        # проверка корректности значения уровеня счастья в созданном экземпляре
        assert dog.happiness == 50
```

Метод `__init__` протестирован.

3. Протестировать остальные методы в классе

Кроме `__init__`, в классе есть метод, который увеличивает уровень счастья собачки. Его тоже нужно покрыть тестами.

Для этого создай объект, вызови для него метод и проверь, что уровень счастья и правда повысился:

```
def test_pet_the_dog_true(self):

        dog = Dog('Sharik')  # создание нового экземпляра класса
        dog.pet_the_dog()    # вызов метода, увеличивающего уровень счастья собачки на 10 процентов

        assert dog.happiness == 60  # проверка, что уровень счастья равен 60-ти процентам (50 + 10)
```

Вот и всё. Класс протестирован.

Тестовое покрытие класса

Как и для метода, для класса можно посчитать тестовое покрытие. Формула расчёта такая:

> 📢 **TestCoverage = (TestCode/TotalCode) * 100%**

Здесь:

- **TestCoverage —** тестовое покрытие кода;
- **TestCode —** строки кода, которые покрыты тестами;
- **TotalCode —** сколько всего строк кода.

Нужно посчитать, сколько в коде строк:

```
class Dog:                    #1

    def __init__(self, name):  #2
        self.name = name       #3
        self.happiness = 50    #4

    def pet_the_dog(self):     #5
        self.happiness += 10   #6
```

Всего получается шесть. Дальше нужно определить, какие строки затрагиваются тестами:

- Первую строку кода проверяют все три теста.
- Вторую строку проверяют тесты также все три теста.
- Третью строку кода проверяет тест `test_name_of_dog_true`.
- Четвертую строку кода проверяет тест `test_default_value_happiness_true`.
- Пятую и шестую строку кода проверяет тест `test_pet_the_dog_true`.

Выходит, тестовое покрытие класса — 100%.

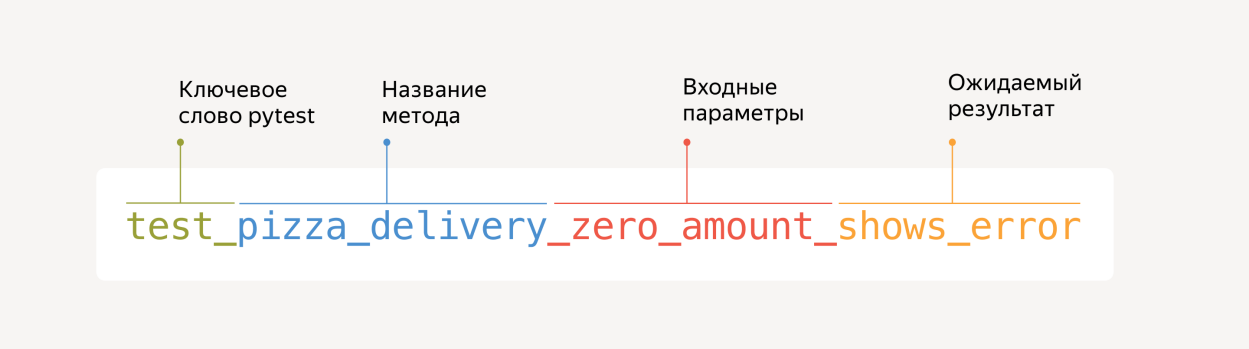
Как называть тесты в pytest

Для юнит-тестов

Названия теста строится так:

1. **Ключевое слово фреймворка pytest** — `test_`. Именно благодаря ему pytest сможет запустить код.
2. **Имя метода, который тестируешь.**
3. **Входные параметры тестирования.**
4. **Ожидаемый результат**.

Например, тестируется метод `pizza_delivery`. Входной параметр — это количество пицц = 0: `zero_amount`. Ожидаемый результат: если пицц 0, тест выдаст ошибку — `shows_error`.



Пиши ожидаемый результат конкретно:

✅ Например, `shows_error`. Или `balance_increased`, если должен пополниться баланс.

❌ Не стоит писать `works_correct` или `everything_is_ok` — непонятно, что именно значит «правильно работает».

Для UI-тестов

📎 Всё как в названии юнит-тестов, но вместо названия метода → название функциональности.

Схема для названия теста:

1. **Ключевое слово фреймворка pytest** — `test`.
2. **Название функциональности, которую тестируешь.**
3. **Входные параметры тестирования.**
4. **Ожидаемый результат**.

Когда пишешь юниты, название метода видно в коде. А вот для функциональности автоматизатор даёт название самостоятельно. Например, если тестируешь кошелёк — `wallet`, если заказ такси — `order_taxi`. Можно посоветоваться с командой и узнать, как будет корректнее и удобнее.

На что обратить внимание

Пиши ожидаемый результат конкретно — так же, как и с юнит-тестами.

✅ `shows_error` или `product_added_to_cart`, если продукт должен добавиться в корзину.

❌ `works_incorrect` или `everything_is_ok`: непонятно, что именно это значит.

Уточняй схему названия тестов у коллег. В каждой компании могут быть свои правила для названий. Схема выше — самая распространённая, но

Фикстуры

Часто бывает, что несколько автотестов в проекте используют одинаковые методы или данные. Получается, в нескольких автотестах лежат одинаковые кусочки кода.

Чтобы упростить жизнь и не дублировать код, придумали **фикстуры**.

Что такое фикстуры

Фикстура — это функция, которая выполнится перед каждым тестом. Например, создаст пользователя или подключится к базе данных.

Пример. Надо проверить два поля объекта «книга» — название и автор. Первый тест проверяет название, второй — автора. Перед каждой проверкой придётся создавать тестовый объект:

```
import pytest

from book import Book

def test_book_name(): # первый тест — проверяет имя
    book = Book(name='Консервный ряд', author='Джон Стейнбейк')
    # код теста

def test_book_author(): # второй тест — проверяет автора
    book = Book(name='Консервный ряд', author='Джон Стейнбейк')
    # код теста
```

Если тестов будет десять, получится десять одинаковых строчек кода. Вместо этого можно написать фикстуру, которая создаёт книгу:

```
import pytest

from book import Book

@pytest.fixture # фикстура, которая создаёт книгу
def book():
    book = Book(name='Консервный ряд', author="Джон Стейнбейк")
    return book
```

Тогда создавать объект и прописывать для него параметры в каждом тесте не нужно:

```
from book import Book

def test_book_name(book):# в тесте не надо создавать объект
    assert book.name == 'Консервный ряд'

def test_book_author(book):# в тесте не надо создавать объект
    assert book.author == 'Джон Стейнбейк'
    # код теста
```

Как создать фикстуру

✅ Чтобы создать фикстуру:

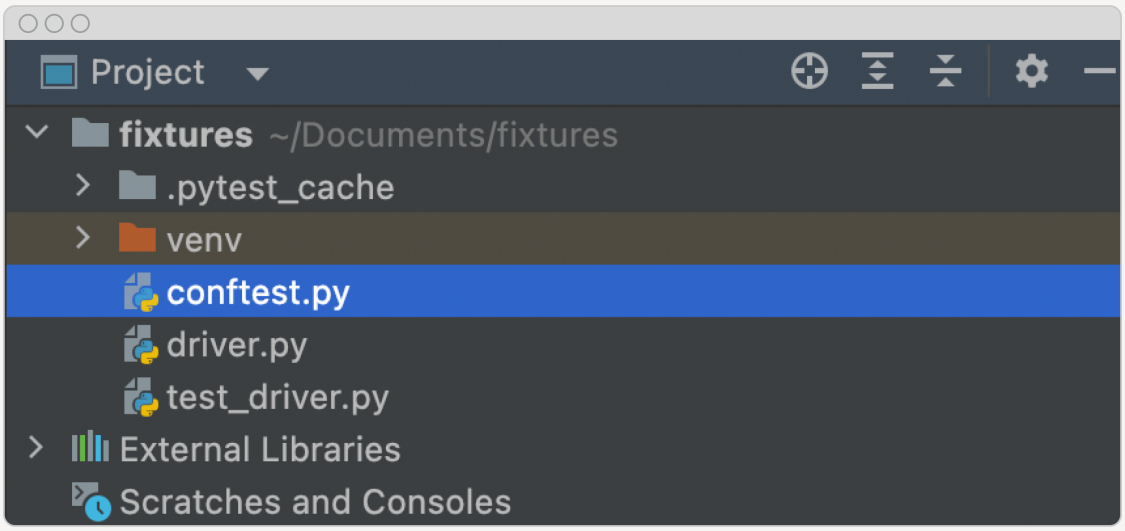
1. Создай файл, где будут храниться все фикстуры
2. Создай функцию
3. Навесь на функцию декоратор

1. Создать файл `conftest.py`

Даже если в проекте одна фикстура, её хранят в отдельном файле. Для pytest файл для фикстур называется `conftest.py`.

Называть его нужно именно так, иначе программа его не найдёт.

Файл хранится в корне проекта.



2. Создать функцию

Фикстура — это функция. Поэтому и создавать её нужно как функцию: начать с ключевого слова `def`, а после написать название.

Название должно соответствовать тому, что фикстура делает. Например:

- `mongo_db_connection` — подключается к базе данных с названием Mongo DB;
- `customer` — создаёт экземпляр класса «покупатель»;
- `wallets` — создаёт набор тестовых кошельков.

3. Навесить декоратор

На фикстуру нужно навесить **декоратор** `@pytest.fixture`. Он и делает из функции фикстуру.

Ещё нужно импортировать библиотеку pytest. Так выглядит файл `conftest.py` с одной фикстурой:

```
import pytest

# класс Company, в котором реализован конструктор и методы
from company import Company

@pytest.fixture # фикстура, которая создаёт компанию
def company():
    company = Company(name='Amazing customer service', manager='Зоя Вексельштейн')

    return company
```

Теперь эту фикстуру можно использовать в файле с тестом `test_company.py`:

```
def test_company_name(company):
    manager = 'Зоя Вексельштейн'
    assert company.manager == manager
```

Когда pytest прочитает этот код, он:

- увидит `company` как аргумент функции и поймёт, что тут замешана фикстура;
- пойдёт искать её в файл `conftest.py`.

Особенности работы с фикстурами

Сколько угодно фикстур в файле

В файле `conftest.py` можно создать сколько угодно фикстур. Располагать их можно в любом порядке: это не влияет на то, как выполняется код с тестами.

Не нужно импортировать файл `conftest.py` напрямую

Допустим, есть файл с тестами `test_hosts.py`. Не надо в нём импортировать `conftest` или фикстуры оттуда. Вот так неправильно:

```
import conftest  # так делать не надо
from conftest import tv_hosts # и так тоже
```

Если так написать, использовать фикстуру не получится. Достаточно просто обращаться к ней в коде теста.

В файле нужно хранить только актуальные фикстуры

Сначала Pytest запускает фикстуры, а потом код самих автотестов. Это может занимать время, особенно если фикстур много.

Практические выводы из этого такие:

- не нужно писать фикстуры «про запас»;
- лучше время от времени удалять фикстуры, которые ты уже не используешь в автотестах.

Один тест может использовать несколько фикстур

Если фикстур несколько, их нужно перечислить по одной через запятую. Например, вот так:

```
def test_check_home_page(postgres_db_connection, popup_data, user)
# postgres_db_connection, popup_data, user — это три разные фикстуры
```

Параметр `scope`

Этот параметр указывает, в каком порядке и как часто вызываются фикстуры. Например:

- перед каждым тестом,
- один раз перед всеми тестами класса,
- один раз перед запуском всех тестов.

Всё зависит от того, какое значение ты задашь параметру. Например, так фикстура выполнится один раз перед каждым классом:

```
@pytest.fixture(scope='class') # задали значение в скобках
```

Значение `'function'`

Фикстура запустится один раз перед каждым тестом, на который её навесили.

Для каждого теста объекты в фикстуре создаются заново, с нуля. Например, тестировщик навесил фикстуру на все тесты класса для проверки авторизации. Для первого теста фикстура создала объект — логин. Допустим, в процессе теста логин поменялся. Для второго теста логин создастся заново: изменения не сохранятся.

Значение `'function'` установлено по умолчанию. Когда тебе приходилось работать с фикстурами в прошлых уроках, там уже был параметр `scope`. Просто когда его не указывают, подразумевается, что значение — `'function'`.

```
@pytest.fixture(scope='function')
@pytest.fixture()
#Это одно и то же. Когда параметр не указан, его значение — function
```

А вот если хочешь задать другое, надо уже прописать это явно.

Значение `'session'`

Фикстура выполнится один раз перед всеми тестами сразу.

```
@pytest.fixture(scope='session')
#фикстура выполнится один раз перед запуском тестов
```

Если какой-то тест изменил значение объекта в фикстуре, остальные будут работать с новым значением. Например, логин изменился с `login` на `login1` — остальные тесты будут использовать `login1`.

> 👉 На практике чаще всего используют фикстуры с `'function'` и `'session'`. Есть значения, которые используют реже. Например, `'class'`.

Значение `'class'`

Фикстура запускается один раз перед тестовым классом, на который её навесят.

Если объект из фикстуры изменился в одном тесте в рамках класса, во всех следующих тестах этого класса будет использоваться измененный объект.

Это удобно, когда тестовые классы строго разделены по функционалу. Например, ты проверяешь страницу авторизации. Может быть так, что `login` поменяется несколько раз. Тестировщик может использовать некорректный логин, чтобы проверить: сервис выдаёт ошибку входа. А для класса проверки карточки сотрудника нужен корректный `login` . Вот фикстура и создаст его заново.