Шпаргалка: юнит-тесты

Что такое юнит-тесты

Юнит-тесты по-другому называют модульными тестами.

Модуль — это целостная часть системы, которая выполняет отдельную функцию. Её можно протестировать изолированно от других.

> 👉 В рабочих задачах юнит-тесты проверяют, что небольшая часть функциональности не сломалась.

Модули в коде

Модули в коде — это отдельные классы и методы.

Пример. Вот программа, которая создаёт собак:

```
class Dog():
    tail = 1
    paws = 4

    def __init__(self, name, color):
        self.dog_name = name
        self.dog_color = color

dog_1 = Dog('Барбос','Чёрный')
dog_2 = Dog('Шарик', 'Коричневый')
dog_3 = Dog('Тедди', 'Рыжий')
```

Здесь в классе `Dog` только один метод — `__init__`. Он и будет модулем.

Зачем нужны юнит-тесты

Если один модуль сломается, это отразится на всей системе. Поэтому прежде чем тестировать программу целиком, проверяют её модули — проводят **юнит-тестирование**.

Юнит-тесты помогают:

- найти, какой именно модуль сломался**.** Если тест не прошёл в одном модуле — значит, ошибка внутри него;
- проверить, что новая функциональность не приводит к ошибкам в уже существующей**.** Если не прошли только тесты на новую функциональность, значит, старая работает корректно;
- выявить ошибки до релиза новой функциональности**.** Если юнит-тесты в новом фрагменте кода не проходят, его нужно дорабатывать.

> 👉 Юнит-тесты проводят, когда функциональность ещё не разработана до конца. Это помогает найти ошибки как можно раньше.

Пирамида тестирования: виды тестов

Чтобы проверить все уровни приложения одних юнит-тестов будет мало. Понадобятся автотесты нескольких видов.

Интеграционные тесты

Они проверяют, как взаимодействуют части системы.

Например, нужно убедиться, что в корзине корректно отображаются цены товаров. Тогда проверяют, как работают вместе два компонента: корзина и отображение цен товаров.

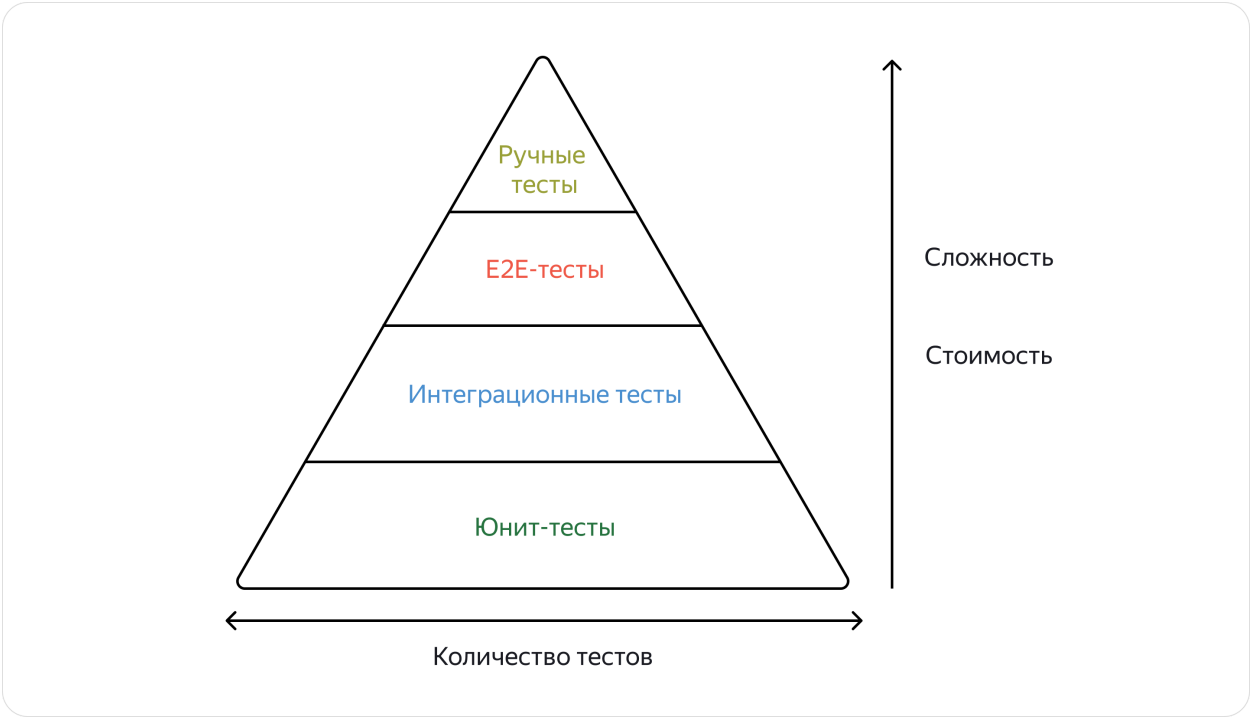
E2E-тесты

Они проверяют работу приложения в целом.

Тесты этого уровня имитируют действия пользователя в виде сценария. Например, пользователь заходит на сайт, регистрируется, ищет товар, заказывает и оплачивает. E2E-тест проделает всё то же самое.

Пирамида тестирования

Все виды тестов объединены в **пирамиде тестирования**. Она показывает, каких тестов в проекте должно быть больше, а каких — меньше:



Внизу — быстрые, надёжные и дешёвые в разработке юнит-тесты: таких тестов должно быть как можно больше. Чем выше тесты в пирамиде, тем они дороже и дольше разрабатываются и выполняются.

👉 Каждый уровень пирамиды отвечает за свою часть тестового покрытия. Только со всеми уровнями продукт будет протестирован до конца.

Базовые правила юнит-тестов

Этот вид тестов пишут по определённым правилам:

- Один тест — одна проверка.
- Независимость данных.
- Независимость тестов.

Правило №1: Один тест — одна проверка

В одном тесте лучше проверять что-то одно: так проще найти ошибку. В тесте с несколькими проверками не всегда очевидно, где именно тест сломался.

Пример. Система должна определить, совершеннолетний ли пользователь. Нужно протестировать метод `is_adult()`.

```
def is_adult(age): # метод is_adult() принимает аргумент age – возраст пользователя

    adult_age = 18 # установили возраст совершеннолетия
    return age >= adult_age # вернётся результат сравнения аргумента с adult_age
```

Чтобы проверить работу метода, ему нужно передать три значения: возраст > 18, возраст = 18, возраст < 18.

👎 Тест с несколькими проверками

Этот тест проверяет два пункта сразу: с возрастом больше 18 — `age_more_than_18` и меньше 18 — `age_less_than_18`.

```
def is_adult_when_age_is_different():

    age_more_than_18 = 19 # создали переменную для первого теста

    actual_result1 = is_adult(age_more_than_18) # сохранили результат выполнения функции is_adult
    expected_result1 = True # сохранили ожидаемый результат для первого теста

    print("Результат выполнения теста:") # вывод в консоль для читабельности
    print(actual_result1 == expected_result1) # результат сравнения актуального и фактического результата первого теста

    age_less_than_18 = 17 # создали переменную для второго теста

    actual_result2 = is_adult(age_less_than_18) # сохранили результат выполнения функции is_adult в переменную

    expected_result2 = False # сохранили ожидаемый результат для второго теста

    print("Результат выполнения теста:")
    print(actual_result2 == expected_result2) # результат сравнения актуального и фактического результата второго теста
```

Если он упадёт, будет сложной найти причину ошибки.

👍 Тест с одной проверкой

Корректный тест для метода `is_adult()` содержит только одну проверку. Выглядит так:

```
def is_adult_when_age_is_more_than18_true():

    age = 25  # задали переменную, которая передаётся в метод как аргумент
    actual_result = is_adult(age)  # вызвали метод, который будем тестировать,
    # передали ему переменную age, а актуальный результат его работы запишется
    # в переменную actual_result
    expected_result = True  # ожидаемый результат проверки — true:
    # мы передали в метод age = 25, а 25 > 18
    print("Результат выполнения теста:")  # сообщение-подсказка для пользователя
    print(actual_result == expected_result)  # сообщение с результатом тестирования:
    # если фактический и ожидаемый результаты совпали, тест считается пройденным
```

Правило №2: Независимость данных

Результаты тестирования одних методов нельзя включать в другие юнит-тесты. Если так сделать, тесты и результаты будут зависеть друг от друга.

Пример. Метод `get_age(name)` принимает параметр `name` — имя пользователя, а из базы данных возвращается его возраст `age`.

👎 Тест, где результат тестирования одного метода передаются в другой

```
def is_adult_when_age_is_more_than18_true():

    name = "Иван" # задали имя пользователя
    age = get_age(name) # передали имя пользователя в метод в качестве
    # параметра и получили его возраст
    actual_result = is_adult(age)
    expected_result = True
    print("Результат выполнения теста:")
    print(actual_result == expected_result)
```

Если в методе `get_age(name)` есть баг, он будет возвращать неправильный возраст или ошибку. Тогда тест `is_adult_when_age_is_more_than18_true()` не будет проходить проверку, хотя проблема не в нём.

👍 Тест, где данные заданы вручную

Чтобы не допустить перехода ошибки, данные для теста нужно задавать вручную — например, объявив переменную.

```
def is_adult_when_age_is_more_than18_true():

    age = 25  # зададим самостоятельно переменную, при помощи которой
    # получим результат из метода isAdult()
    actual_result = is_adult(age)  # передадим эту переменную в метод
    expected_result = True
    print("Результат выполнения теста:")
    print(actual_result == expected_result)
```

Правило №3: Независимость тестов

Важно, чтобы тесты были независимы друг от друга. Это значит, их можно запускать по одному и в любом порядке.

Пример. В игре есть корзина с яблоками. Есть два теста: один проверяет добавление яблок в корзину, а другой — удаление из корзины.

```
class BasketWithApple():   # создали класс корзины с яблоками

    apples_in_basket = 0   # переменная класса

    def get_quantity_of_apples(self):  # создали метод, который возвращает количество яблко в корзине
        return self.apples_in_basket

    def add_apple_to_basket(self):     # метод добавляет яблоко в корзину
        self.apples_in_basket = self.apples_in_basket + 1

    def delete_apple_from_basket(self):  # метод удаляет яблоко из корзины
        self.apples_in_basket = self.apples_in_basket - 1
```

👎 Взаимосвязанные тесты

Первый тест кладёт яблоко в корзину и проверяет, что яблоко добавилось. Второй достаёт именно это яблоко и проверяет, что яблок больше нет.

```
def test_add_and_delete_apple():  # тест, который проверяет и добавление, и удаление яблока из одной корзины

    busket_with_apple = BasketWithApple()  # создание экземпляра класса в рамках теста
    busket_with_apple.add_apple_to_basket()  # вызов метода добавления яблока в корзину
    print(busket_with_apple.get_quantity_of_apples() == 1)  # проверка, что корзина содержит одно яблоко
    busket_with_apple.delete_apple_from_basket()  # вызов метода удаления яблока из корзины
    print(busket_with_apple.get_quantity_of_apples() == 0)  # проверка, что корзина не содержит яблок
```

👍 Независимые тесты

Здесь у каждого теста — своя корзина с яблоками, и каждый проверяет фактический результат согласно документации.

```
def test_add_apple_to_basket(): # создание теста, который проверяет добавление яблока в корзину

    basket_with_apple = BasketWithApple() # создание экземпляра класса в рамках теста
    basket_with_apple.add_apple_to_basket() # вызов метода добавления яблока в корзину
    print(basket_with_apple.get_quantity_of_apples() == 1) # проверка, что корзина содержит одно яблоко


def test_delete_apple_from_basket(): # создание теста, который проверяет удаление яблока из корзины

    basket_with_apple = BasketWithApple() # создание экземпляра класса в рамках теста
    basket_with_apple.add_apple_to_basket() # вызвали метод, который добавляет яблоко в корзину
    basket_with_apple.delete_apple_from_basket() # вызвали метод, который удаляет яблоко из корзины
    print(basket_with_apple.get_quantity_of_apples() == 0) # проверка, что корзина не содержит яблок
```

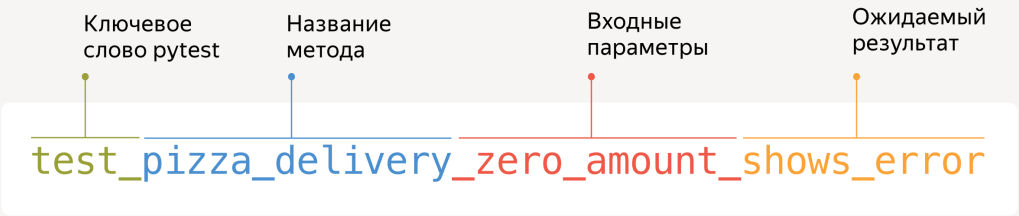
Этот сценарий позволяет не привязываться к результатам предыдущего теста при разработке новых, а также дает выбор при запуске тестов.

Как называть юнит-тесты в pytest

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.

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

Чтобы описать точнее, смотри в документацию. Если там написано, что при правильной работе метод выводит true, так и нужно написать: `shows_true`.