

# Контроль качества программного кода

## Цель занятия

После освоения темы:

- Вы узнаете о роли тестирования в обеспечении качества ПО и познакомитесь с различными видами тестирования.
- Научитесь анализировать код статически.
- Научитесь создавать автоматические тесты.
- Научитесь измерять покрытие кода тестами.

## План занятия

1. [Обеспечение качества и тестирование ПО](#)
2. [Инструменты статического анализа кода](#)
3. [Инструменты тестирования](#)
4. [Использование фикстур и мок-объектов в PyTest](#)
5. [Покрытие кода тестами. Плагин pytest-cov](#)

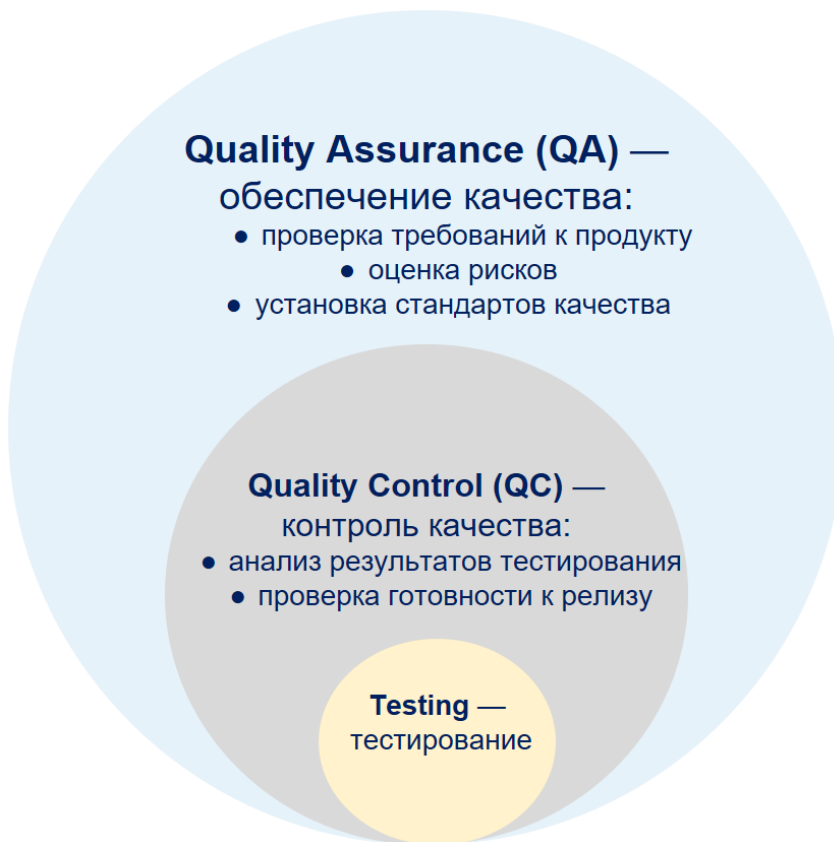
## Конспект занятия

### 1. Обеспечение качества и тестирование ПО

#### Как обеспечивается качество программных продуктов

Чтобы выпустить продукт с надлежащими характеристиками, необходим ряд мероприятий. Обобщенно они называются **обеспечением качества** и настроены на

выстраивание процесса таким образом, чтобы гарантировать соблюдение требований заказчика и принятых стандартов качества.



Неотъемлемый компонент этого процесса — **контроль качества**. Он направлен на то, чтобы минимизировать вероятность дефектов той версии программного продукта, которую получит заказчик.

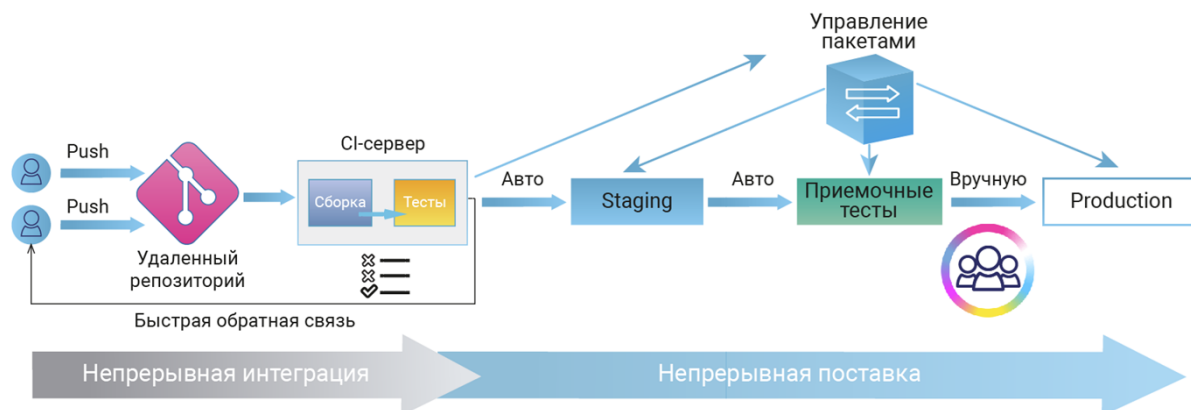
Тестирование — главный компонент контроля качества.

### Тестирование в процессе выпуска версий продукта

Тестирование играет важную роль на всех стадиях подготовки продукта к выпуску, начиная с первых шагов разработки и заканчивая приемочными испытаниями. Причем это справедливо для любых методологий разработки.

Возьмем популярную концепцию **непрерывной поставки**. Основная ее идея заключается в том, чтобы доставлять изменения конечным пользователям

максимально быстро. Релизы при этом могут выпускаться часто, хоть каждый день. Можно представить, насколько такой процесс подвержен возникновению ошибок.



**Пример.** Программист допустил опечатку и в операторе сравнения вместо знака «>» поставил знак «≥». После подключения новой версии библиотеки окажется, что какая-то функция больше не доступна для новой версии программы.

Большинство таких ошибок можно избежать при помощи тестов, в том числе автоматических.

## Как разработать необходимые тесты

В общем случае нужно проверить реакцию системы или конкретного ее компонента на различные входные данные. Проверка всех возможных комбинаций входных значений невозможна, поэтому есть ряд подходов к проектированию тестовых сценариев. Рассмотрим три распространенных подхода.

### Проектирование тестовых сценариев:

- Эквивалентное разделение.
- Анализ граничных значений.
- Таблицы принятия решений.

**Эквивалентное разделение.** Пусть некоторая функция принимает на вход значения от 1 до 10.

Для сценария понадобятся:

- позитивный тест — одно верное значение внутри интервала (например, 5);
- негативный тест — одно неверное значение вне интервала (например, 0).

**Анализ граничных значений.** Пусть допустимые значения от 1 до 10.

Для сценария понадобятся:

- позитивный тест — минимальная и максимальная границы (1 и 10);
- негативный тест — значения больше и меньше границ (0 и 11).

Эквивалентное разделение и анализ граничных значений лучше комбинировать, но в целом всегда надо ориентироваться на здравый смысл и точно понимать, что мы хотим проверить.

Для более сложных систем, например, для системы определения продажи страховки для автомобиля, целесообразно построить **таблицу принятия решений**:

Определение страховки авто	Сценарий развития событий							
Условия (входящие параметры)	1	2	3	4	5	6	7	8
Число происшествий с авто > X	Да	Да	Да	Нет	Нет	Нет	Нет	Нет
Марка авто в списке {..., ..., ...}	Да	Да	Нет	Нет	Да	Да	Нет	Нет
Возраст авто > Y	Да	Нет	Нет	Нет	Да	Нет	Да	Нет
Результат действия								
- Сообщение "в страховке отказано" - Возврат на главную страницу	✓	✓	✓					
- Сообщение "Страховка одобрена" - Установить стандартную ставку - Сформировать чек - Отобразить информацию				✓				
- Сообщение "Страховка одобрена" - Установить ставку со скидкой - Сформировать чек - Отобразить информацию					✓	✓	✓	✓

Составлять такие таблицы — трудоемкая задача. Обычно тестирование на основе таких сценариев в основном выполняется вручную.

## Классификация работ по тестированию ПО

- Ручное и автоматическое тестирование.
- Статическое и динамическое тестирование.

**Статическое тестирование** — тестирование программного кода без его фактического выполнения. С его помощью можно выявлять такие проблемы, как объявленная переменная, которая нигде не используется, дублирование названий и т. д.

**Динамическое тестирование** — выполнение программы в тестовой среде с целью найти в ней ошибки.

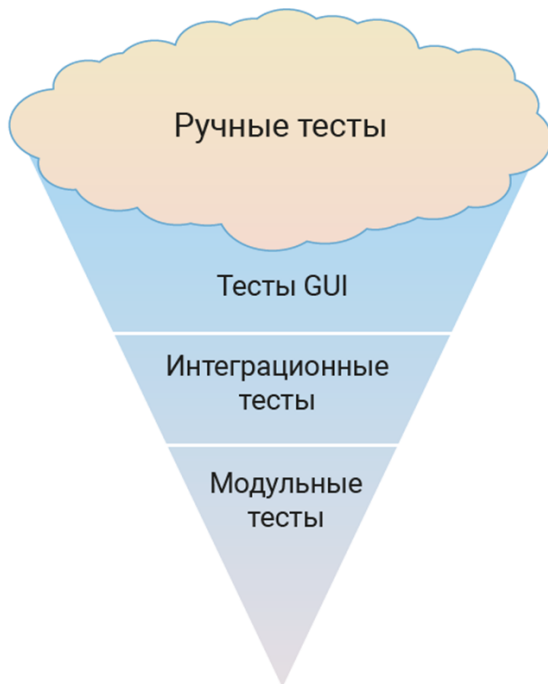
- Функциональное (модульное, интеграционное) и нефункциональное тестирование (UI/UX, производительность, безопасность). При **функциональном тестировании** проводится тестирование отдельных компонентов системы (классы, функции) изолированно. **Интеграционное тестирование** рассматривает систему в целом, то есть проверяет связи между компонентами, их совместную работу.

## Пирамида тестирования

Пирамида тестирования построена по принципу того, какие тесты более дешевые, а какие — более дорогие; какие тесты разрабатывать проще, а какие — сложнее.



Такая схема предлагается как некий паттерн, но нужно иметь в виду, что во многих организациях этот паттерн превращается в антипаттерн:



От ручного тестирования отказаться невозможно. Тем не менее не стоит забывать, что нужно стремиться автоматизировать большую часть процессов.

## 2. Инструменты статического анализа кода

Рассмотрим один из инструментов статического анализа кода — линтер `pylint`.

**Линтер** — утилита, которая проверяет, что код «хорошо» написан. То есть написан с соблюдением определенных требований стиля и не содержит явных ошибок.

Установим утилиту `pylint` с помощью `pipenv`:

```
$ pipenv install pylint
```

Далее активируем виртуальное окружение:

```
$ pipenv shell
```

Запускаем утилиту `pylint` и передаем ей файл для анализа:

```
$ pylint some\ file.py
```

Утилита выдает общую оценку кода, и выводит список ошибок в стиле написания. Ошибки имеют код обозначения, буква в начале кода указывает на вид ошибки:

- W (warning) — предупреждение;
- C (convention) — нарушение соглашений;
- R (refactor) — рефакторинг;
- E (error) — ошибка.

Утилита `pylint` позволяет оставить в коде комментарии, чтобы настроить проверку.

Рассмотрим еще одну утилиту — `mypy`. Эта утилита проверяет соответствие типов в программе, если аннотации типов указаны.

Установим библиотеку `mypy`:

```
$ pipenv install mypy
```

Активировав новое виртуальное окружение, мы можем воспользоваться утилитой. Мы можем настроить `mypy` так, чтобы она проверяла весь проект, но пока сосредоточимся на проверке одного файла:

```
$ mypy student.py
```

Проверка всего проекта может занять много времени, поскольку утилита анализирует всю кодовую базу со всеми зависимостями. Для одного файла утилита сработает достаточно быстро.

В результате утилита покажет список всех ошибок с указанием строк, в которой данная ошибка содержится. Утилита проводит глубокий анализ кода, который позволяет избежать многих потенциальных ошибок. Но чтобы утилита была полезна, необходимо снабжать код аннотациями типов.

### 3. Инструменты тестирования

Написание тестов к коду — трудная рутинная работа. Но она может гарантировать качество кода.

В стандартной библиотеке Python есть модуль, который позволяет писать тесты к коду непосредственно в документации, — модуль `doctest`. Подробный разбор примера модуля можно посмотреть в [документации](#). Чтобы посмотреть работу модуля, в конце файла есть строки:

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

После запуска файла ничего не появится. И это правильно, `doctest` прогоняет тесты, и если все сработало верно, никак это не отображает. Если будет ошибка — программа об этом сообщит. Если появится ошибка, незадокументированная в модуле, программа также сообщит об ошибке.

Работа модуля настраивается с помощью специальных комментариев:

```
# doctest: +NORMALIZE_WHITESPACE
```

заставляет проигнорировать лишние пробелы и прочитать данные вне зависимости от того, как расставлены пробелы.

```
# doctest: +ELLIPSIS
```

позволяет вставлять многоточие

В сложных проектах удобнее использовать библиотеку `pytest`. В тестирующий код импортируется нужная функция и модуль `pytest`. Далее мы пишем некоторые функции — все функции, которые начинаются со слова `test_`, библиотека `pytest` будет запускать. Сам файл также должен начинаться с `test_`.

Внутри каждой функции с помощью инструкции `assert` мы прописываем, что ожидаем получить.

**Важно!** В каждой тестирующей функции рекомендуется проверять что-то одно. Чаще всего имеет смысл проверить крайние случаи.

Чтобы выполнить тесты, необходимо зайти в терминал и запустить `pytest` без дополнительных параметров. Библиотека сама проанализирует каталог и увидит, что есть файлы с именем `test_`, и внутри них функции с `test_`. Утилита сама выполнит тесты. Утилита должна быть предварительно установлена.

## 4. Использование фикстур и мок-объектов в PyTest

### Мок-объекты

Проиллюстрируем использование мок-объектов на простейшем примере:



```
from include.mymodule import get_lucky_number

def my_function():
    number = get_lucky_number()
    return number

if __name__=="__main__":
    print("Вызываем функцию my_function...")
    print(my_function())
```

Здесь функция зависит от другой функции, которая импортируется из некоторого модуля:

```
import random

def get_lucky_number():
    return random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Предположим, что это не просто случайное число, а некоторое сложное вычисление, которое может занимать определенное время:

```
import random
import time

def get_lucky_number():
    time.sleep(2)
    return random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

При выполнении программы эта задержка заметна.

Мы хотим протестировать работу функции `my_function`. Для этого нужно разработать тест. Создаем папку `tests` в корне проекта, а затем новый файл `test_my_function.py`, в котором напомним тест для нашей функции:

```
from main import my_function

def test_my_function():

    lucky_number = my_function()

    assert lucky_number in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Теперь создаем пустой файл `__init__.py` в папке с тестами.

В командной строке запускаем PyTest. Видим задержку в 2 секунды, как и при выполнении основной программы. Это неприемлемо для тестирования, особенно если таких долгих процедур в программе много.

Запросы, которые занимают много времени:

- запрос к базе данных;
- запрос к внешнему сервису.

Если мы хотим протестировать определенные компоненты изолированно, мы не хотим зависеть от внешних сущностей. Здесь и приходят на помощь мок-объекты. В нашем случае можно заменить на мок-объект импортируемую функцию, которая выдает случайно одно из 10 чисел.

**Мок-объект** — это некоторый фиктивный объект, на который можно заменить зависимость в тестируемом объекте.

Для PyTest есть удобная вспомогательная библиотека, которая позволяет реализовывать мок-объекты. Установим ее:

```
pip install pytest-mock
```

Теперь сделаем замену на мок:

```
from main import my_function

def test_my_function(mock):

    mock.patch(

        'main.get_lucky_number',
```

```
        return_value = 5 #возвращаем фиктивное значение
    )

    lucky_number = my_function()

    assert lucky_number in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

И теперь тест выполняется за доли секунды.

Рассмотрим еще один пример, в котором тестируемая функция будет зависеть не от другой функции, а от класса:

```
import random
import time

def get_lucky_number():
    time.sleep(2)
    return random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

class DataProvider():
    def __init__(self):
        self.data = {}

    def get_data(self):
        time.sleep(2)
        self.data = {"field1": 100, "field2": 200}
        return self.data
```

Зададим еще одну функцию, которая будет зависеть от класса:

```
from include.mymodule import get_lucky_number, DataProvider

def my_function():
    number = get_lucky_number()
    return number

def my_function_1():
    provider = DataProvider()
```

```
data = provider.get_data()
return data

if __name__=="__main__":
    print("Вызываем функцию my_function...")
    print(my_function())
    print("Вызываем функцию my_function_1...")
    print(my_function_1())
```

И снова видим ожидаемые задержки между выполнениями функций.

Посмотрим, как можно заменить метод из класса `DataProvider` фиктивным методом. Вначале напишем обычный тест, убедимся, что он работает 2 секунды:

```
from main import my_function, my_function_1
def test_my_function(mock):
    mock.patch(
        'main.get_lucky_number',
        return_value = 5 #возвращаем фиктивное значение
    )

    lucky_number = my_function()
    assert lucky_number in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def test_my_function_1(mock):
    data = my_function_1()
    assert data['field1'] == 100
    assert data['field2'] == 200
```

Воспользуемся методом `patch`:

```
from main import my_function, my_function_1
def test_my_function(mock):
    mock.patch(
```

```
        'main.get_lucky_number',
        return_value = 5 #возвращаем фиктивное значение
    )

    lucky_number = my_function()
    assert lucky_number in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def test_my_function_1(mocker):
    def mock_get_data(self):
        return {"field1": 100, "field2": 200}
    mocker.patch(
        'main.DataProvider.get_data',
        mock_get_data
    )
    data = my_function_1()
    assert data['field1'] == 100
    assert data['field2'] == 200
```

Таким образом, удалось заменить долгий процесс на фиктивный объект, благодаря чему тест выполняется быстро.

Итак, мок-объектами можно подменять зависимости от внешних объектов для эффективной работы тестов.

## Фикстуры

Фикстуры нужны, чтобы обеспечивать тесты наборами из известных данных. Типичным примером фикстуры будет набор тестовых учетных записей пользователей для проверки системы авторизации.

Можно создать фикстуру один раз, а затем использовать ее в разных тестах.

Если посмотреть на наши тесты, то в примерах можно применить фикстуру [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] для порождения набора чисел:

```
from main import my_function, my_function_1
import pytest

@pytest.fixture #для фикстуры пользуемся декоратором
def lucky_numbers():
    return [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def test_my_function(mocker, lucky_numbers):
    mocker.patch(
        'main.get_lucky_number',
        return_value = 5 #возвращаем фиктивное значение
    )

    lucky_number = my_function()
    assert lucky_number in lucky_numbers

def test_my_function_1(mocker):
    def mock_get_data(self):
        return {"field1": 100, "field2": 200}

    mocker.patch(
        'main.DataProvider.get_data',
        mock_get_data
    )

    data = my_function_1()
    assert data['field1'] == 100
    assert data['field2'] == 200
```

Итак, фикстуры позволяют получать наборы тестовых данных для многократного использования, тем самым избегая дублирования кода в тестах.

## 5. Покрытие кода тестами. Плагин pytest-cov

**Покрытие кода тестами** — это показатель того, какая часть программного кода охвачена тестами. Он позволяет оценить качество набора тестов в проекте.

Чтобы измерить покрытие кода тестами при использовании библиотеки PyTest, необходимо дополнительно установить библиотеку `pytest-cov`:

```
pip install pytest-cov
```

После установки этой библиотеки можно указать аргумент `cov` для измерения покрытия кода:

```
pytest --cov
```

```
(.venv) user@user-Lenovo-ideapad-S530-13IWL:~/project$ pytest --cov
===== test session starts =====
platform linux -- Python 3.11.6, pytest-7.4.3, pluggy-1.3.0
rootdir: /home/user/project
plugins: mock-3.12.0, cov-4.1.0
collected 2 items

tests/test_my_function.py .. [100%]

----- coverage: platform linux, python 3.11.6-final-0 -----
Name                               Stmts  Miss  Cover
-----
include/mymodule.py                 12     5    58%
main.py                             13     4    69%
tests/__init__.py                    0     0   100%
tests/test_my_function.py           16     0   100%
-----
TOTAL                                41     9    78%
```

В выводе утилиты видно, что два файла с кодом `mymodule` и `main` покрыты тестами на 58 и 69%. Чтобы сгенерировать детальный отчет о покрытии, можно воспользоваться ключом `cov-report`:

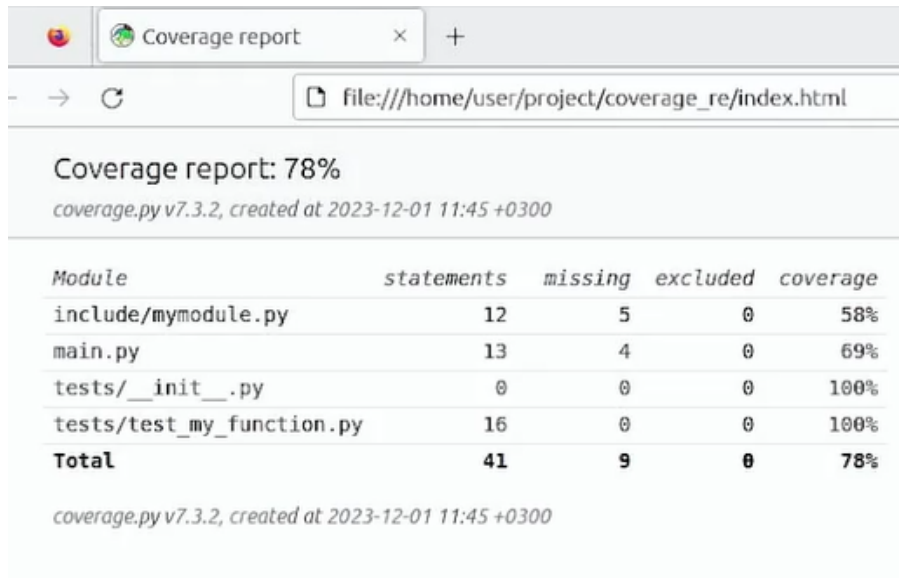
```
pytest --cov --cov-report=html:coverage_re
```

```
(.venv) user@user-Lenovo-ideapad-S530-13IWL:~/project$ pytest --cov --cov-report=html:coverage_re
===== test session starts =====
platform linux -- Python 3.11.6, pytest-7.4.3, pluggy-1.3.0
rootdir: /home/user/project
plugins: mock-3.12.0, cov-4.1.0
collected 2 items

tests/test_my_function.py .. [100%]

----- coverage: platform linux, python 3.11.6-final-0 -----
Coverage HTML written to dir coverage_re
```

По такой команде будет создана папка `coverage_re`, в которой появится HTML-файл. Он показывает информацию о покрытии.



Module	statements	missing	excluded	coverage
include/mymodule.py	12	5	0	58%
main.py	13	4	0	69%
tests/__init__.py	0	0	0	100%
tests/test_my_function.py	16	0	0	100%
<b>Total</b>	<b>41</b>	<b>9</b>	<b>0</b>	<b>78%</b>

Откроем файл в браузере и посмотрим, почему получилось 58% покрытия:



## Coverage report: 78%

coverage.py v7.3.2, created at 2023-12-01 11:45 +0300

Module	statements	missing	excluded	coverage
include/mymodule.py	12	5	0	58%
main.py	13	4	0	69%
tests/__init__.py	0	0	0	100%
tests/test_my_function.py	16	0	0	100%
<b>Total</b>	<b>41</b>	<b>9</b>	<b>0</b>	<b>78%</b>

coverage.py v7.3.2, created at 2023-12-01 11:45 +0300

## Coverage for include/mymodule.py: 58%

12 statements 7 run 5 missing 0 excluded

« prev ^ index » next coverage.py v7.3.2, created at 2023-12-01 11:45 +0300

```
1 import random
2 import time
3
4
5 def get_lucky_number():
6     time.sleep(2)
7     return random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
8
9
10 class DataProvider():
11     def __init__(self):
12         self.data = {}
13
14     def get_data(self):
15         time.sleep(2)
16         self.data = {"field1": 100, "field2": 200}
17         return self.data
```

« prev ^ index » next coverage.py v7.3.2, created at 2023-12-01 11:45 +0300

Здесь подсвечены те инструкции, в которые тестирующая система не заходила при запуске тестов. В файле `mymodule` не были протестированы функции `get_lucky_number` и `get_data`. Если мы хотим достичь 100% покрытия, то необходимо написать тесты для этих двух функций.

Создадим файл `test_mymodule.py`, в который поместим тесты для обозначенных функций:

```
from include.mymodule import get_lucky_number

from include.mymodule import DataProvider

def test_get_lucky_number():

    assert get_lucky_number() in [1, 2, 3, 4, 5, 6, 7, 8, 9,
    10]

def test_get_data():

    provider = DataProvider()

    data = provider.get_data()

    assert data['field1'] == 100

    assert data['fiels2'] == 200
```

Убедимся, что тесты проходят:

```
(.venv) user@user-Lenovo-ideapad-S530-13IWL:~/project$ pytest
===== test session starts =====
platform linux -- Python 3.11.6, pytest-7.4.3, pluggy-1.3.0
rootdir: /home/user/project
plugins: mock-3.12.0, cov-4.1.0
collected 4 items

tests/test_my_function.py .. [ 50%]
tests/test_mymodule.py .. [100%]
```

```
===== 4 passed in 4.02s =====
(.venv) user@user-Lenovo-ideapad-S530-13IWL:~/project$ pytest --cov --cov-report=html:coverage_re
===== test session starts =====
platform linux -- Python 3.11.6, pytest-7.4.3, pluggy-1.3.0
rootdir: /home/user/project
plugins: mock-3.12.0, cov-4.1.0
collected 4 items

tests/test_my_function.py .. [ 50%]
tests/test_mymodule.py .. [100%]

----- coverage: platform linux, python 3.11.6-final-0 -----
Coverage HTML written to dir coverage_re
```

И откроем наш отчет заново:

## Coverage report: 92%

coverage.py v7.3.2, created at 2023-12-01 11:56 +0300

Module	statements	missing	excluded	coverage
include/mymodule.py	12	0	0	100%
main.py	13	4	0	69%
tests/__init__.py	0	0	0	100%
tests/test_my_function.py	16	0	0	100%
tests/test_mymodule.py	9	0	0	100%
<b>Total</b>	<b>50</b>	<b>4</b>	<b>0</b>	<b>92%</b>

coverage.py v7.3.2, created at 2023-12-01 11:56 +0300

Теперь видим, что для файла `mymodule` покрытие составляет 100%.

Стоит обратить внимание, что не всегда целесообразно гнаться за 100% покрытием. В нашем примере мы вынужденно протестировали инструкции `time.sleep(2)`, чтобы достичь показателя 100%. Если мы уверены, что строка не содержит каких-либо проблем, можно исключить ту или иную инструкцию из анализа покрытия. Продемонстрируем это на примере основного файла:

```
from include.mymodule import get_lucky_number, DataProvider

def my_function():
    number = get_lucky_number()
    return number

def my_function_1():
    provider = DataProvider()
    data = provider.get_data()
    return data

if __name__ == "__main__":
    print("Вызываем функцию my_function...") # pragma: no cover
    print(my_function())
    print("Вызываем функцию my_function_1...")
    print(my_function_1())
```

При включении комментария pragma: no cover модуль pytest-cov будет его игнорировать.

## Coverage report: 94%

coverage.py v7.3.2, created at 2023-12-01 12:02 +0300

Module	statements	missing	excluded	coverage
include/mymodule.py	12	0	0	100%
main.py	12	3	1	75%
tests/__init__.py	0	0	0	100%
tests/test_my_function.py	16	0	0	100%
tests/test_mymodule.py	9	0	0	100%
<b>Total</b>	<b>49</b>	<b>3</b>	<b>1</b>	<b>94%</b>

coverage.py v7.3.2, created at 2023-12-01 12:02 +0300

## Coverage for main.py: 75%

12 statements 9 run 3 missing 1 excluded

« prev ^ index » next coverage.py v7.3.2, created at 2023-12-01 11:59 +0300

```

1 | from include.mymodule import get_lucky_number, DataProvider
2 |
3 |
4 | def my_function():
5 |     number = get_lucky_number()
6 |     return number
7 |
8 | def my_function_1():
9 |     provider = DataProvider()
10 |    data = provider.get_data()
11 |    return data
12 |
13 |
14 | if __name__ == "__main__":
15 |    print("Вызываем функцию my_function...") # pragma: no cover
16 |    print(my_function())
17 |    print("Вызываем функцию my_function_1...")
18 |    print(my_function_1())
19 |

```

« prev ^ index » next coverage.py v7.3.2, created at 2023-12-01 11:59 +0300

## Дополнительные материалы для самостоятельного изучения

1. [The CI/CD principles/ <Packt>](#)
2. [Тест-анализ и тест-дизайн / Mellarius](#)