

# Python

## Занятие #10. Инструменты тестирования в python

 **О чем это занятие:**

- ☐ Уровни тестирования
- ☐ Простейшие подходы
- ☐ Фреймворки
- ☐ Дополнительные инструменты

 **Теги:**

модульное тестирование, интеграционное тестирование, фреймворки, doctest, assert, unittest, pytest, coverage.py

## Краткий обзор темы

- ❑ Уровни тестирования, на которых остановимся: модульное, интеграционное, системное
- ❑ Простейшие подходы в тестировании:
  - вывод результата с помощью **print** – легко реализуемо, но высока вероятность не отследить ошибки
  - **doctest** – вызов и ожидаемый результат, описанные в коде внутри тройных кавычек. Подходит для функций, результат которых легко представим в виде строки
  - **assert** – в код функции добавляется проверка результата, при несовпадении генерируется исключение. Легко читается и пишется, но сообщения об ошибках приходится писать самостоятельно
- ❑ Тестирование с помощью фреймворков:
  - **unittest** входит в состав стандартной библиотеки, сам находит тесты, но имеет нехарактерную для Python архитектуру и синтаксис, тесты описываются довольно объемным кодом
  - **pytest** требует установки, но наиболее универсален, запускает тесты, написанные как в своем синтаксисе, так и для unittest и в doctest
- ❑ Дополнительные инструменты тестирования: unittest.mock, coverage.py

# Зачем писать тесты?

Тестирование программного обеспечения – проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом.

## Плюсы использования тестов:

- ❑ При помощи тестов осуществляется проверка корректности кода
- ❑ Проверка работоспособности кода после внесения изменений (безопасный рефакторинг)
- ❑ Тесты сами по себе являются примерами использования кода, некоторой документацией к функции/библиотеке

## Минусы использования тестов:

- ❑ Написание тестов требует времени и ресурсов
- ❑ Во многих случаях тестов приходится писать больше, чем кода, который они тестируют
- ❑ Наличие тестов всё равно не гарантирует корректность работы функции

**!** Для объемных и сложно устроенных функций/библиотек тесты необходимы.

# Уровни тестирования

Существует пять уровней тестирования, рассмотрим три первых уровня:

## ❑ Модульное тестирование (Unit Testing)

Модульное тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по отдельности (модули программ, объекты, классы, функции и т.д.)

## ❑ Интеграционное тестирование (Integration Testing)

Интеграционное тестирование проверяет взаимодействие между компонентами системы после проведения модульного тестирования

## ❑ Системное тестирование (System Testing)

Системное тестирование проверяет работу проекта в целом.

 <https://habr.com/post/279535/> – Тестирование. Фундаментальная теория

## Пример: term\_frequency

```
1 from typing import Iterator, Sequence, Tuple, Hashable
2 from collections import Counter
3
4
5 def term_frequency(sequence: Sequence[Hashable]) -> Iterator[Tuple]:
6     term_cnt = len(sequence)
7     for term, frequent in Counter(sequence).items():
8         yield (term, round(frequent / term_cnt, 3))
9
10
11 terms = ['Moscow', 'New York', 'Moscow', 'London']
12 list(term_frequency(terms))
13
14
15 Out: [('Moscow', 0.5), ('New York', 0.25), ('London', 0.25)]
```

Здесь импортируется базовый класс последовательности Sequence, создается функция term\_frequency для подсчета частоты появления слов в тексте, в качестве входного параметра функция принимает любой хэшируемый объект, в примере это список.

## Простейшие подходы

### print

Самый простой способ протестировать работу функции, это включить в код вывод результата с помощью оператора `print`:

```
1 def test_tf() -> None:
2     terms = ['Moscow', 'New York', 'Moscow', 'London']
3     print(list(term_frequency(terms)))
4
Out: [('Moscow', 0.5), ('New York', 0.25), ('London', 0.25)]
```

Для простых функций, поведение которых легко предсказать, такое тестирование, возможно, наименее трудоемкое, но:

- ❑ этот подход реализует визуальную проверку результата
- ❑ при использовании `print` легко не заметить ошибку

**✗ Не стоит тестировать код с помощью print!**

## Doctest

```
1 """
2 Compute tf of each term in the sequence
3
4 >>> list(term_frequency([]))
5 []
6 >>> list(term_frequency('MNMLPTNPMBV'))
7 [('M', 0.273), ('N', 0.182), ('L', 0.091), ('P', 0.182), ('T', 0.091),
8  ('B', 0.091), ('V', 0.091)]
9 """
```

👉 doctest пишется в docstring, т.е. в тройных кавычках сразу под определением функции. После тройного символа “>>>” записывается код, который необходимо выполнить и строчкой ниже результат, который должен быть получен. Эти тесты сами по себе являются документацией к функции.

🔍 <https://docs.python.org/3/library/doctest.html> – Test interactive Python examples

```

1 from typing import Iterator, Sequence, Tuple, Hashable
2 from collections import Counter
3 import doctest
4
5
6 def term_frequency(sequence: Sequence[Hashable]) -> Iterator[Tuple]:
7     """
8     Compute tf of each term in the sequence
9
10    >>> list(term_frequency([]))
11    []
12    >>> list(term_frequency('MNMLPTNPMBV'))
13    [('M', 0.273), ('N', 0.182), ('L', 0.091), ('P', 0.182), ('T', 0.091),
14     ('B', 0.091), ('V', 0.091)]
15    """
16    term_cnt = len(sequence)
17    for term, frequent in Counter(sequence).items():
18        yield (term, round(frequent / term_cnt, 3))
19
20
21 if __name__ == '__main__':
22     doctest.testmod()

```

Здесь показано, как записываются doctest для функции term\_frequency.

При вызове doctest.testmod() будут показаны все doctest, описанные в текущем файле.



```

1 $ python ./tf_doctest.py # or python -m doctest tf_doctest.py
2 *****
3 File "./tf_doctest.py", line 12, in __main__.term_frequency
4 Failed example:
5     list(term_frequency('MNMLPTNPMBV'))
6 Expected:
7     [('M', 0.273), ('N', 0.182), ('L', 0.091), ('P', 0.182), ('T', 0.091),
8     ('B', 0.091), ('V', 0.091)]
9 Got:
10    [('M', 0.273), ('N', 0.182), ('L', 0.091), ('P', 0.182), ('T', 0.091),
11    ('B', 0.091), ('V', 0.091)]
12 *****
13 1 items had failures:
14   1 of 2 in __main__.term_frequency
15 ***Test Failed*** 1 failures.

```

Здесь показан случай, когда тест из doctest не проходит, хотя визуально строки одинаковые. Это происходит потому, что в doctest ожидаемый результат содержит символ переноса строки, которого нет в реальном результате.

👉 Такой итог сравнения говорит о том, что Python сравнивает строки из doctest и реального результата.

Поведение doctest можно контролировать:

- ❑ с помощью флагов: -o NORMALIZE\_WHITESPACE
- ❑ с помощью директив: # doctest: +ELLIPSIS

```

1 $ python -m doctest -v -o NORMALIZE_WHITESPACE tf_doctest.py
2 Trying:
3     list(term_frequency('MNMLPTNPMBV'))
4 Expecting:
5     [('M', 0.273), ('N', 0.182), ('L', 0.091), ('P', 0.182), ('T', 0.091),
6      ('B', 0.091), ('V', 0.091)]
7 ok
8 1 items passed all tests:
9   2 tests in tf_doctest.term_frequency
10 2 tests in 2 items.
11 2 passed and 0 failed.
12 Test passed.

```

NORMALIZE\_WHITESPACE не различает последовательности пробелов и переносов строк.

```

1 """
2 Compute tf of each term in the sequence
3
4 >>> list(term_frequency([]))
5 []
6 >>> list(term_frequency('MNMLPTNPMBV')) # doctest: +ELLIPSIS
7 [('M', 0.273), ('N', 0.182)...('B', 0.091), ('V', 0.091)]
8 """

```

ELLIPSIS – троеточие  
(...) соответствует любой подстроке

#### ❑ Плюсы doctest:

- Доступен в стандартной библиотеке Python
- Решает задачу тестирования для небольших проектов
- Тесты довольно просты в реализации и восприятии
- Примеры кода в документации всегда актуальны

#### ❑ Минусы doctest:

- Требует содержательного строкового представления от тестов (не всегда реализуемо, особенно при операциях с изменяемыми адресами или для функций с неявным `return`)
- Длинные примеры кода ухудшают читаемость документации
- Нет способа запустить подмножество тестов (например, выборочные тесты)

## assert

Оператор `assert` принимает два аргумента:

- ❑ условие
- ❑ произвольное значение

Если условие ложно, то поднимается исключение **AssertionError**

```
1 def test_tf():
2     assert term_frequency('MNMN') == [('M', 0.5), ('N', 0.5)]
3
Out: Traceback (most recent call last):
  File "tf_assert.py", line 16, in <module>
    test_tf()
  File "tf_assert.py", line 12, in test_tf
    assert term_frequency('MNMN') == [('M', 0.5), ('N', 0.5)]
AssertionError
```

Здесь описана функция `test_tf`, содержащая `assert`. В этом операторе прописано условие, которое должно выполняться, чтобы тест считался пройденным.

💬 Тест не прошёл, т.к. `term_frequency` выдает генератор, а не список, а в `assert` её результат сравнивается со списком. В предыдущем примере мы использовали `list(term_frequency)`.

```
1 def test_tf():
2     actual = term_frequency('MNMN')
3     expected = [('M', 0.5), ('N', 0.5)]
4     err_msg = f'{actual} != {expected}'
5     assert actual == expected, err_msg
6
7 Out: Traceback (most recent call last):
   File "tf_assert02.py", line 20, in <module>
     test_tf()
   File "tf_assert02.py", line 16, in test_tf
     assert actual == expected, err_msg
AssertionError: <generator object term_frequency at 0x1035cfc00> != [('M', 0.5), ('N', 0.5)]
```

Здесь описана переменная `actual` для функции, `expected` – для ожидаемого результата, `err_msg` – для сообщения об ошибке, для понимания природы ошибки.

В выводе видим, что в исключении `AssertionError` слева стоит генератор, а справа – список.

```

1 def test_empty():
2     actual = list(term_frequency(''))
3     expected = []
4     err_msg = f'{actual} != {expected}'
5     assert actual == expected, err_msg
6
7 def test_contain_one_term():
8     actual = list(term_frequency('MMMMNNNPPL'))
9     expected = ('L', 0.1)
10    err_msg = f'{actual} not contain {expected}'
11    assert expected in actual, err_msg

```

Здесь описано ещё два теста, обрабатывающих другие случаи входных данных.

#### ❑ Плюсы использования assert:

- Доступен в стандартной библиотеке, т.к. это часть языка
- Легко читать
- Пишется в виде обычных функций

#### ❑ Минусы использования assert:

- Необходимость запуска вручную
- Дублирование формирования err\_msg
- Дублирование кода проверки на равенство
- Необходимость самостоятельного формирования сообщения об ошибке

## Фреймворки

**Фреймворк** (в переводе с англ. framework – каркас) – это набор библиотек для автоматизации рутинных действий, внесения в процесс разработки большей предсказуемости и комфорта, упрощения связи между разными частями приложения.

Рассмотрим два примера фреймворка, используемых для тестирования кода в Python.

### Unittest

Модуль unittest реализует функциональность JUnit (из Java) для тестирования кода на Python

💬 Наследие Java до сих пор присутствует в API в виде названий методов и классов и архитектуры тестов

 <https://docs.python.org/3/library/unittest.html> – Unit testing framework

```

1 import unittest
2
3
4 class TestTF(unittest.TestCase):
5     def test_tf(self):
6         actual = term_frequency('MNMN')
7         expected = [('M', 0.5), ('N', 0.5)]
8
9         self.assertEqual(actual, expected)
10
11     def test_empty(self):
12         actual = list(term_frequency(''))
13         expected = []
14
15         self.assertEqual(actual, expected)
16
17 if __name__ == '__main__':
18     unittest.main()

```

Здесь описаны два теста `test_tf` и `test_empty`, в каждом вызван тип исключения `assertEqual`, выбрасываемый при несовпадении ожидаемого и реального результатов. В конце кода вызывается выполнение всего файла с помощью `unittest.main()`.



```

1 $ python test_tf_unittest01.py
2 .F
3 =====
4 FAIL: test_tf (__main__.TestTF)
5 -----
6 Traceback (most recent call last):
7   File "test_tf_unittest01.py", line 17, in test_tf
8     self.assertEqual(actual, expected)
9 AssertionError: <generator object term_frequency at 0x108d1c4f8> !=
10 [('M', 0.5), ('N', 0.5)]
11 -----
12 Ran 2 tests in 0.000s
13
14
15 FAILED (failures=1)

```

Здесь в первой строке идет запуск тестов, во второй строке точка обозначает, что первый тест прошёл, а F – что второй не прошёл, получен Fail. Сообщение об ошибке в AssertionError генерируется автоматически.

👉 unittest.main загружает и запускает все тесты текущего модуля

👉 unittest может сам находить и запускать тесты

💬 для этого файлы должны начинаться со слова test, класс должен наследоваться от TestCase

По сравнению с предыдущим примером поменялась первая строка:

```

1 $ python -m unittest
2 .F
3 =====
4 FAIL: test_tf (test_tf_unittest02.TestTF)
5 -----
6 Traceback (most recent call last):
7   File "...test_tf_unittest02.py", line 17, in test_tf
8     self.assertEqual(actual, expected)
9 AssertionError: <generator object at 0x103a13930> != [('M', 0.5), ('N', 0.5)]
10
11 -----
12 Ran 2 tests in 0.001s
13
14 FAILED (failures=1)

```

С помощью unittest возможен запуск одного теста ( в данном случае тест прошел, о чем говорит точка во второй строке):

```

1 $ python -m unittest test_tf_module.TestTF.test_empty
2 .
3 -----
4 Ran 1 test in 0.000s
5
6 OK

```

👉 Методы для проверки, включенные в unittest:

```
1 self.assertEqual(a, b) # a == b
2 self.assertNotEqual(a, b) # a != b
3 self.assertTrue(x) # bool(x) is True
4 self.assertFalse(x) # bool(x) is False
5 self.assertIs(a, b) # a is b
6 self.assertIsNot(a, b) # a is not b
7 self.assertIsNone(x) # x is None
8 self.assertIsNotNone(x) # x is not None
9 self.assertIn(a, b) # a in b
10 self.assertNotIn(a, b) # a not in b
11 self.assertIsInstance(a, b) # isinstance(a, b)
12 self.assertNotIsInstance(a, b) # not isinstance(a, b)
13 ...
14 self.assertRaises(Exception)
```

👉 unittest содержит методы для подготовки контекста: `setUp`, `tearDown`, `setUpClass`, `tearDownClass`.

- ❑ Метод `setUpClass` запускается перед выполнением всех тестов набора, метод `setUp` – перед выполнением каждого отдельного теста.
- ❑ Метод `tearDown` запускается после окончания каждого теста (выполнение либо сообщение об ошибке), а метод `tearDownClass` – после прохождения всех тестов набора.

```

1 class TestTF(unittest.TestCase):
2     def setUp(self):
3         self.text = open('./text', 'w+')
4
5     def tearDown(self):
6         self.text.truncate(0)
7         self.text.close()
8
9     def test_from_file(self):
10        self.text.write('M\nN\nM\nN')
11        self.text.flush()
12        self.text.seek(0)
13
14        actual = list(term_frequency(self.text.read().splitlines()))
15        self.assertEqual(actual, [('M', 0.5), ('N', 0.5)])

```

Здесь описан один тест `test_from_file` и подготовлен контекст, т.к. описаны методы `setUp` и `tearDown`, в которых рабочий файл открывается на чтение и закрывается после работы.

#### ❑ Плюсы unittest:

- Доступен в стандартной библиотеке
- Понятные сообщения об ошибках
- Автоматически находит тесты

#### ❑ Минусы unittest:

- Не PEP8
- Многословный
- Низкая читаемость

## Pytest

- Популярная альтернатива unittest для написания и запуска тестов
- Практически отсутствуют специфичные интерфейсы – используются стандартные средства языка

 <https://docs.pytest.org/en/latest/contents.html> – Full pytest documentation

❑ Установка пакета с помощью pip:

```
1 $ pip install -U pytest
```

❑ Тесты выглядят следующим образом:

```
1 def test_tf():
2     assert term_frequency('MNMN') == [('M', 0.5), ('N', 0.5)]
3
4
5 def test_empty():
6     assert list(term_frequency('MNMN')) == []
```

❑ Запуск теста (в первой строке) и вывод сообщения об ошибке от pytest:

```

1 $ python -m pytest
2 ===== test session starts =====
3 platform darwin -- Python 3.7.1, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
4 rootdir: ../tf_pytest, inifile:
5 collected 2 items
6
7 test_tf.py F. [100%]
8 ===== FAILURES =====
9 _____ test_tf _____
10
11     def test_tf():
12 >         assert term_frequency('MNMN') == [('M', 0.5), ('N', 0.5)]
13 E         AssertionError: assert <generator ob...t 0x107c9f138> == [('M', 0.5), ('N', 0.5)]
14 E             Use -v to get the full diff
15
16 test_tf.py:5: AssertionError
17 ===== 1 failed, 1 passed in 0.08 seconds =====

```

❑ Запуск одного теста из набора (в первой строке) и вывод сообщения о том, что тест пройден:

```
1 $ python -m pytest test_tf.py::test_empty
2 ===== test session starts =====
3 platform darwin -- Python 3.7.1, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
4 rootdir: ../tf_pytest, inifile:
5 collected 1 item
6
7 test_tf.py . [100%]
8
9 ===== 1 passed in 0.05 seconds =====
```

❏ Как pytest осуществляет поиск тестов:

- pytest запускает все файлы вида test\_\*.py или \*\_test.py
- будут запущены все функции, начинающиеся с test\_\*
- также будут запущены тесты, написанные с использованием unittest
- и, наконец, будут запущены doctest, если поставить флаг --doctest-modules

👉 pytest – это удобный инструмент для объединения различных тестов и их автоматического запуска.

#### ❑ [Интроспекция:](#)

```
1  _____ test_in_range _____
2
3  def test_in_range():
4      x = 23
5  >   assert x in range(10)
6  E   assert 23 in range(0, 10)
7  E   + where range(0, 10) = range(10)
8
9  test_tf.py:14: AssertionError
10 ===== 1 failed, 2 passed in 0.05 seconds =====
```

Здесь мы видим, что pytest показывает, чему равно значение параметра x в момент сравнения

#### ❑ Проверка поднятия исключения:

```
1  import pytest
2
3
4  def test_exception():
5      py_tools = {'print': False, 'doctest': True, 'assert': False,
6                  'unittest': False, 'pytest': True}
7
8      with pytest.raises(KeyError):
9          py_tools['jUnit']
```

Здесь проверяется, выбрасывается ли исключение при возникновении соответствующего случая.



## ❏ Параметрические тесты

```
1 import pytest
2
3
4 def capitalize(s: str) -> str:
5     return s.capitalize()
6
7
8 @pytest.mark.parametrize('s,exp', [
9     ('moscow', 'Moscow'),
10    ('london', 'London'),
11    ('paris', 'paris')
12 ])
13 def test_capitalize(s, exp):
14     assert capitalize(s) == exp
```

Для написания параметрического теста нужно использовать декоратор `@pytest.mark.parametrize`, в скобках первым аргументом указываются те параметры, которые затем встретятся в тестах (в данном случае 's,exp'), вторым аргументом — список тестируемых данных.

Далее функция `test_capitalize` протестирует все наборы входных данных из списка декоратора.

❑ Подготовка контекста с fixture:

```
1 @pytest.fixture()
2 def text():
3     text_file = open('./text', 'w+')
4     yield text_file
5     text_file.truncate(0)
6     text_file.close()
7
8
9 def test_from_file(text):
10     text.write('M\nN\nM\nN')
11     text.flush()
12     text.seek(0)
13
14     actual = list(term_frequency(text.read().splitlines()))
15
16     assert actual == [('M', 0.5), ('N', 0.5)]
```

Код внутри функции `text` до инструкции `yield` выполняется до запуска тестов, так, как это было в методе `setUp` в `unittest`. Код после инструкции `yield` выполняется после запуска тестов (аналогично методу `tearDown` из `unittest`).

## ❑ встроенные контексты

```
1 def make_banana():
2     cnt = int(input())
3     print(' '.join(['banana'] * cnt))
4
5
6 def test_make_banana(capsys, monkeypatch):
7     handle = io.StringIO('3')
8     monkeypatch.setattr(sys, 'stdin', handle)
9     make_banana()
10    captured = capsys.readouterr()
11    assert captured.out[:-1] == 'banana banana banana'
```

Здесь описана функция `make_banana`, в которой нет аргументов и нет `return`, поэтому её сложно тестировать и показан способ тестирования с помощью `monkeypatch`. В `captured.out[:-1]` возвращаемся на один символ назад, т.к. в сравниваемой строке не нужен символ переноса строки.

## ❑ Плюсы pytest

- Тесты выглядят как обычные функции
- Понятные сообщения об ошибках
- Отличная документация
- Механизм параметризации тестов
- Автоматически находит тесты
- Фикстуры, которые можно переиспользовать и делать композиции
- Много дополнительных возможностей и расширений

## ❑ Минусы pytest

- Сложная реализация pytest
- Отличный от JUnit подход к тестированию

## Дополнительные инструменты тестирования

### unittest.mock

- ❑ Позволяет заменить части тестируемого кода на mock-объекты
- ❑ Предоставляет данные об использовании замененных объектов

 <https://docs.python.org/3/library/unittest.mock.html> – mock object library

Проверка параметров при вызове:

```
1 from unittest.mock import MagicMock
2
3 from pyavito import Avito
4
5
6 avito = Avito('access_token')
7 avito.append_ads = MagicMock()
8 avito.append_ads(title='iPhone', price='₽23000')
9
10 avito.append_ads.assert_called_with(title='iPhone', price='₽23000')
```

Здесь показано, как тестируются методы модуля pyavito. MagicMock не только фиксирует вызовы функции, но и может эмулировать, что вернет конкретный вызов.

patch и установка возвращаемого значения:

```
1 from unittest.mock import patch
2
3 from pyavito import Avito
4
5
6 def test_get_ads():
7     exp_ads = dict(id=4, title='iPhone', price='₽23000')
8
9     with patch.object(Avito, 'get_ads', return_value=exp_ads):
10         avito = Avito('access_token')
11         ads = avito.get_ads(exp_ads['id'])
12
13     assert ads == exp_ads
```

Patching позволяет заменить реализацию и вернуть нужное ожидаемое значение.

## coverage.py

Дополнительный пакет, требует установки с помощью pip.

- ❑ Следит, какой код запускался в тестах, а какой нет
- ❑ Предоставляет отчеты в различных форматах: html, xml и др.

 <https://coverage.readthedocs.io> – coverage.py

 <https://pytest-cov.readthedocs.io/> – pytest-cov's documentation

### Пример использования coverage.py.

```
1 from typing import Sequence
2
3
4 def calc_discount(weekday: int, shopping_list: Sequence[str]) -> int:
5     saturday_index = 5
6     max_discount = 10
7     discount_per_unit = 1
8
9     if weekday < saturday_index:
10         return 0
11     else:
12         return min(max_discount, len(shopping_list) * discount_per_unit)
```

Здесь описана  
функция calc\_discount.

```
1 import calendar
2
3 from .discount import calc_discount
4
5
6 def test_discount_in_weekend():
7     assert calc_discount(calendar.SATURDAY, ['apple', 'orange']) == 2
```

Здесь описан тест для  
функции  
test\_discount\_in\_weekend.

```

1 $ python -m pytest --cov .
2 ===== test session starts =====
3 collected 1 item
4
5 test_discount.py . [100%]
6
7 ----- coverage: platform darwin, python 3.7.1-final-0 -----
8 Name                Stmts   Miss  Cover
9 -----
10 __init__.py           0      0   100%
11 discount.py           8      1    88%
12 test_discount.py      4      0   100%
13 -----
14 TOTAL                 12      1    92%
15
16 ===== 1 passed in 0.05 seconds =====

```

Здесь произведен  
замер покрытия  
тестом функции,  
иначе говоря, все ли  
строки кода в  
функции  
запускаются и  
отрабатывают.

coverage.py позволяет выгрузить данные о покрытии тестами в виде отчета в html:

```

1 $ python -m pytest --cov . --cov-report html

```

## Coverage report: 92%

| <i>Module ↓</i>               | <i>statements</i> | <i>missing</i> | <i>excluded</i> | <i>coverage</i> |
|-------------------------------|-------------------|----------------|-----------------|-----------------|
| <code>__init__.py</code>      | 0                 | 0              | 0               | 100%            |
| <code>discount.py</code>      | 8                 | 1              | 0               | 88%             |
| <code>test_discount.py</code> | 4                 | 0              | 0               | 100%            |
| <b>Total</b>                  | <b>12</b>         | <b>1</b>       | <b>0</b>        | <b>92%</b>      |

Отчет в html по отдельно взятой функции:


## Coverage for **discount.py** : 88%


8 statements 7 run 1 missing 0 excluded

```
1 | from typing import Sequence
2 |
3 |
4 | def calc_discount(weekday: int, shopping_list: Sequence[str]) -> int:
5 |     saturday_index = 5
6 |     max_discount = 10
7 |     discount_per_unit = 0.1
8 |
9 |     if weekday < saturday_index:
10 |         return 0
11 |     else:
12 |         return min(max_discount, len(shopping_list) * discount_per_unit)
```



## Полезные ссылки

 <https://hypothesis.readthedocs.io/> – генерирует различные тестовые данные

 <https://factoryboy.readthedocs.io/> – упрощает создание тестовых данных

 [http://developer.paylogic.com/articles/factory-injection-combining-pytest-factory\\_boy.html](http://developer.paylogic.com/articles/factory-injection-combining-pytest-factory_boy.html) – использование factoryboy и pytest