

Библиотеки unittest и pytest

- 1 Повторение: зачем нужны библиотеки для тестирования
- 2 Библиотека unittest
- 3 Библиотека pytest
- 4 Общие рекомендации о том, как писать тесты
- 5 Резюме
- 6 Бонус. Библиотека doctest

Аннотация

В этом уроке мы рассмотрим инструменты языка Python для тестирования: библиотеки unittest и pytest. С их помощью мы научимся писать гибкие и автоматизированные тесты.

1. Повторение: зачем нужны библиотеки для тестирования

На прошлом уроке мы узнали, что такое юнит-тестирование. Оказалось, что тестировать свои программы очень полезно, но не так уж и просто:

- **Тесты должны быть гибкими.** Нужно тестировать самые разные случаи: не только правильность возвращаемых значений, но и, например, исключения
- **Тесты должны быть простыми.** Если код теста сложен, то в нем легко ошибиться. К тому же, тесты — это еще и неявный способ документирования кода
- **Тесты должны быть автоматизированными.** Запуск ста тестов должен быть таким же простым, как и запуск одного
- Отчет о непройденных тестах должен быть удобным и подробным, чтобы легко понять, где именно ошибка

Сегодня мы познакомимся с библиотеками, которые позволяют писать гибкие, простые и автоматизированные тесты.

2. Библиотека unittest

unittest — стандартная библиотека Python для unit-тестирования. Ей пользуются не очень часто, потому что pytest, речь о которой пойдет чуть ниже, завоевала сердца абсолютного большинства тех, кто пишет тесты

на Python (а к тому же еще она совместима с тестами, которые написаны для unittest), но, тем не менее, было бы несправедливо обойти вниманием этот инструмент.

Давайте узнаем, как устроено тестирование с помощью этой библиотеки на примере функции `reverse()` из предыдущего урока. Поместим функцию в отдельный файл `reverse.py` и подключим его к тестирующему модулю.

```
# Тестируемая функция
def reverse(s):
    if type(s) != str:
        raise TypeError('Expected str, got {}'.format(type(s)))

    return s[::-1]
```

В тестирующем модуле сначала подключим библиотеку `unittest`, а затем из файла `reverse.py` импортируем функцию `reverse`.

```
import unittest
from reverse import reverse
```

Для каждого тестируемого компонента (в нашем случае — функции `reverse()`) нужно реализовать класс-наследник от `unittest.TestCase`. Методы этого класса, название которых начинается с `test_`, и будут тестами. У базового класса `unittest.TestCase` есть встроенные методы для проверки возвращаемых значений — в частности, метод `assertEqual`, который проверяет, соответствует ли полученное значение ожидаемому.

```
class TestReverse(unittest.TestCase):
    def test_empty(self):
        self.assertEqual(reverse(''), '')
```

Обратите внимание, что название метода `test_empty` подсказывает нам, какой именно случай тестируется (пустая строка).

Тестирование исключений через `unittest` делается с помощью метода `assertRaises`. Это менеджер контекста, который принимает на вход один аргумент — ожидаемое исключение (в нашем случае это `TypeError`):

```
def test_wrong_type(self):
    with self.assertRaises(TypeError):
        reverse(42)
```

Автоматизировать тестирование просто: в конце программы нужно дописать

```
if __name__ == '__main__':
    unittest.main()
```

и сохранить весь код в файл с расширением `.py` (например, в `unittest_simple.py`). Запускается так же, как обычная программа на Python.

```
>python files/unittest_simple.py
-----
Ran 2 tests in 0.000s
OK
```

Два теста (мы описали два метода в классе `TestReverse`) прошли успешно.

3. Библиотека pytest

По сравнению с unittest, тесты pytest синтаксически проще. Вот пример такого теста:

```
# Тестируемая функция
def reverse(s):
    if type(s) != str:
        raise TypeError('Expected str, got {}'.format(type(s)))

    return s[::-1]

# Обязательно начинайте тест с префикса 'test_'
def test_reverse():
    assert reverse('abc') == 'cba'
```

Код теста — это обычная функция на Python. Для сравнения возвращаемого значения с ожидаемым используется конструкция `assert`.

Запускается тест из командной строки с помощью специальной утилиты `pytest`, которая устанавливается вместе с библиотекой.

```
pytest files/pytest_simple.py
===== test session starts =====
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /materials/additional/tests, inifile:
collected 1 item

files/pytest_simple.py .

===== 1 passed in 0.01 seconds =====
```

Запуск программы `pytest_simple.py` из консоли с помощью `python` или из вашей IDE ни к чему не приведет. В этой программе нет вызова функций или блока `if __name__ == "__main__":`:

```
python files/pytest_simple.py
```

Утилита `pytest` анализирует код следующим образом:

1. Находит **функции**, названия которых начинаются с `test_`, и выполняет их как тесты
2. Находит **классы**, названия которых начинаются с `Test`. У классов находит **методы**, названия которых начинаются с `test_`, и выполняет их как тесты

Все проверки внутри тестовых функций или методов можно делать с помощью стандартного макроса `assert`. Для проверки исключений в библиотеке `pytest`, по аналогии с библиотекой `unittest`, есть специальный менеджер контекста `pytest.raises()`. Чтобы использовать его, сперва нужно импортировать библиотеку `pytest`.

```
import pytest

def test_exception():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

Для автоматизации тестирования с помощью pytest ничего, в общем-то, делать не нужно — достаточно написать тестовые функции (не забывая, что их название должно начинаться с `test_`). Утилита pytest способна принимать на вход несколько файлов с тестами, так что для каждой компоненты можно писать тесты в отдельном файле.

Разумеется, это далеко не вся функциональность библиотеки pytest, а только ее крошечная часть. pytest позволяет создавать фикстуры (fixtures) — специальные функции, которые выполняются до или после непосредственно вызова теста. С помощью фикстур можно, например, сначала залить в базу данных тестовые данные, а после вызова теста удалить их.

Еще одним мощным инструментом являются моки (Mock). С помощью них можно заменять какие-то части вашей программы при тестировании на макеты, которые обладают некоторым фиксированным поведением. Например, можно заменить реальный сервер системы на имитацию, которая возвращает заранее заготовленный ответ.

4. Общие рекомендации о том, как писать тесты

Какие входные данные нужно тестировать?

1. Тестируйте корректность работы на **неправильных** входных данных. Примеры:

- Неправильный тип аргумента (число вместо строки)
- Некорректное значение (квадратный корень из отрицательного числа или деление на ноль)

В этих случаях программа должна выбрасывать **исключение**. При тестировании нужно проверять, что удаляется нужное исключение.

2. Тестируйте **граничные** случаи. Например:

- Пустая строка
- Пустой список
- Ноль и т. д.

3. Не забывайте протестировать **правильные** входные данные :)

Как организовать код тестов?

1. Для каждого компонента заведите свой тестовый класс или отдельный файл. Названия тестовых функций должны отражать смысл теста:

- Пример плохого названия: `test_1`
- Пример хорошего названия: `test_palindrome`

2. Желательно, чтобы в тестовой функции была ровно одна проверка

- Пример «неправильного» теста функции `foo(n)` на pytest:

```
def test_foo():
    assert foo(0) == 1
    with pytest.raises(TypeError):
        foo('42')
```

- Пример «правильного» теста:

```
def test_foo_zero():
    assert foo(0) == 1

def test_foo_wrong_type():
    with pytest.raises(TypeError):
        foo('42')
```

5. Резюме

1. Мы познакомились с библиотеками unittest и pytest, которые позволяют писать простые, гибкие и автоматизированные тесты
2. Мы изучили общие рекомендации к написанию тестов:
 - Какие данные тестировать: не только «ожидаемые» входные значения, но также неправильные типы и граничные случаи
 - Как организовывать код тестов: отдельный тестовый класс для каждого компонента, одна проверка в одном тесте

6. Бонус. Библиотека doctest

В разговоре про модульное тестирование стоит упомянуть такой инструмент, как doctest. Эта библиотека, хоть и не обладает слишком обширными возможностями, зачастую бывает удобна для создания простых тестов.

Модуль интересен тем, что тесты на нем выглядят как скопированные фрагменты сеанса из интерактивной строки Python.

Давайте рассмотрим пример функции, которая возвращает квадратный корень из числа. Если аргумент не число, тогда бросается исключение `TypeError`, а если аргумент меньше нуля — исключение `ValueError`. С тестами для doctest такая функция будет выглядеть вот так:

```
def square_root(x):
    """
    >>> square_root(4)
    2.0
    >>> [square_root(x) for x in range(10)]
    [0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979, 2.449489742783178,
    >>> square_root(-1)
    Traceback (most recent call last):
        ...
    ValueError: Argument must be greater than 0
    >>> square_root("yandexlyceum")
    Traceback (most recent call last):
        ...
    TypeError: yandexlyceum is not valid int or float value
    """
    if not isinstance(x, int) and not isinstance(x, float):
        raise TypeError(f"{x} is not valid int or float value")
    if x >= 0:
        return x ** 0.5
    else:
        raise ValueError("Argument must be greater than 0")
```

```
if __name__ == "__main__":
    import doctest

    doctest.testmod()
```

Для того чтобы запустить файл в режиме прохождения тестов, надо в командной строке выполнить команду:

```
python имя_файла.py -v
```

Мы получим вот такой результат:

```
Trying:
    square_root(4)
Expecting:
    2.0
ok
Trying:
    [square_root(x) for x in range(10)]
Expecting:
    [0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979, 2.449489742783178,
ok
Trying:
    square_root(-1)
Expecting:
    Traceback (most recent call last):
      ...
    ValueError: Argument must be greater than 0
ok
Trying:
    square_root("yandexlyceum")
Expecting:
    Traceback (most recent call last):
      ...
    TypeError: yandexlyceum is not valid int or float value
ok
1 items had no tests:
    __main__
1 items passed all tests:
   4 tests in __main__.square_root
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

Очевидно, что doctest не подходит для написания сложных тестов, для этого лучше использовать pytest. Но есть сценарии, в которых использование именно этой библиотеки более чем оправдано:

- Написание тестов для проверки базовой функциональности функций и объектов
- Проверка, что документация к функциям правильная и актуальная
- Генерация документации по своему модулю с примерами ввода-вывода

Справка

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках сервиса, принадлежат АНО ДПО «Образовательные технологии Яндекса». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «Образовательные технологии Яндекса».

[Пользовательское соглашение.](#)

© 2018 – 2024 ООО «Яндекс»