Библиотеки unittest и pytest

Перед уроком повторите темы:

— Менеджер контекста **with** (рассматривался на уроке по работе с файлами);

— Команда **assert**.

План урока

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 нём легко ошибиться. К тому же, тесты — это ещё и неявный способ документирования кода;

— **Тесты должны быть автоматизированными.** Запуск ста тестов должен быть таким же простым, как и запуск одного;

— Отчёт 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.

Давайте узнаем, как устроено тестирование с помощью этой библиотеки на примере функции **reverse()** с 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: /Users/ЯндексЛицей/Уроки/materials/additional/2 year/14, 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 приведёт. 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 фукнции (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.

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 массив;

— 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')
```

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.

Помощь