Инструменты для тестирования в Python обширны. Какие-то из них легко внедряются в код, какие-то нет. Сегодня мы познакомимся с новым инструментом - doctest. Этот с натяжкой попадает под категорию - модульное тестирование. Можно применять для тестирования как функций, так и классов.

О doctest

doctest интересен тем, что использование выглядит, как будто пишем код в REPL.

Функция factorial - для вычисления факториала. Использование функции такое:

**>>**> factorial(5)

В результате вызова:

**>>**> factorial(5)

120

И здесь на арену выходит doctest. Модуль ищет фрагменты текста, которые выглядят как интерактивные python сессии. Далее выполняет сеансы и проверяет, совпадает ли с тем что указано в docstring.

В принципе на этом можно и закончить урок - пишем в докстроках код из REPL и тест готов. Doctest прям в чистую выполняет код из докстрок, а значит можно городить сложную логику.

Несколько распространенных способов использования doctest:

* Проверить, что документация к функциям отражает суть и не устарела.
* Для выполнения регрессионного тестирования, убедившись, что интерактивные примеры из тестового файла или тест-объект работает, как ожидалось.
* Писать учебник документации для пакета, обильно иллюстрированный примерами ввода-вывода.

Стоит рассмотреть несколько примеров использования.

Doctest для функций

Для примера рассмотрим такой код:

""*"*

This **is** the "example" module.

The example module supplies one **function**, **factorial**(). **For** **example**,

>>> factorial(5)

120

""*"*

def factorial(n):

""*"Return the factorial of n, an exact integer >= 0.*

If the result **is** small enough **to** fit in **an** **int**, **return** **an** **int**.

Else **return** **a** long.

>>> [factorial(n) **for** n in range(6)]

[1, 1, 2, 6, 24, 120]

>>> [factorial(long(n)) **for** n in range(6)]

[1, 1, 2, 6, 24, 120]

>>> factorial(30)

265252859812191058636308480000000L

>>> factorial(30L)

265252859812191058636308480000000L

>>> factorial(-1)

Traceback (most recent **call** **last**):

...

ValueError: n must **be** >= 0

Factorials of floats are OK, but the float must **be** **an** exact integer:

>>> factorial(30.1)

Traceback (most recent **call** **last**):

...

ValueError: n must **be** exact integer

>>> factorial(30.0)

265252859812191058636308480000000L

It must also not **be** ridiculously large:

>>> factorial(1e100)

Traceback (most recent **call** **last**):

...

OverflowError: n too large

""*"*

import math

**if** not n >= 0:

raise ValueError("n must be >= 0")

**if** math.floor(n) != n:

raise ValueError("n must be exact integer")

**if** n+1 == n: # **catch** **a** value like 1e300

raise OverflowError("n too large")

result = 1

factor = 2

**while** factor <= n:

result \*= factor

factor += 1

**return** result

**if** \_\_name\_\_ == "\_\_main\_\_":

import doctest

doctest.testmod()

Функция вычисляет факториал в цикле. В блоке с docstring заметно и описание функции, и конструкции похожие на строки кода из REPL. Даже не похожие, а прям они! Это основная особенность doctest - прям в документации пишешь тесты. Что может быть проще? А, ну да, надо документацию писать.

В примере простые вызовы функций да и получение исключений.

Честно и без обмана. Чтобы убедится что тесты работают, выполняем:

$ **python** example.**py** -v

Trying:

factorial(5)

Expecting:

120

ok

Trying:

[factorial(n) **for** n in range(6)]

Expecting:

[1, 1, 2, 6, 24, 120]

ok

Trying:

[factorial(long(n)) **for** n in range(6)]

Expecting:

[1, 1, 2, 6, 24, 120]

ok

Trying:

factorial(30)

Expecting:

265252859812191058636308480000000L

ok

Trying:

factorial(30L)

Expecting:

265252859812191058636308480000000L

ok

Trying:

factorial(-1)

Expecting:

Traceback (most recent **call** **last**):

...

ValueError: n must **be** >= 0

ok

Trying:

factorial(30.1)

Expecting:

Traceback (most recent **call** **last**):

...

ValueError: n must **be** exact integer

ok

Trying:

factorial(30.0)

Expecting:

265252859812191058636308480000000L

ok

Trying:

factorial(1e100)

Expecting:

Traceback (most recent **call** **last**):

...

OverflowError: n too large

ok

2 items passed **all** tests:

1 tests in \_\_main\_\_

8 tests in \_\_main\_\_.factorial

9 tests in 2 items.

9 passed and 0 failed.

Test passed.

Тесты пройдены. Функция работает! Использование doctest крайне легкое и это определяет простоту внедрения.

Debug doctest

Дебажить такие тесты совсем просто и делается двумя путями:

* писать код в REPL, что самое лучшее - сразу копируешь код в строки документации.
* воспользоваться функцией doctest.script\_from\_examples:

**import** doctest

**print** doctest.script\_from\_examples(r"""

Set x and y to 1 and 2.

>>> x, y = 1, 2

Print their sum:

>>> print x+y

3

""")

Тесты классов

Тестировать функции легче и редко вызывает сложности. А вот с классами... Рассмотрим класс:

**class** **Test**(**object**):

**def** **\_\_init\_\_**(**self**, number):

**self**.number = number

**def** **multiply\_by\_2**(**self**):

**return** **self**.number\*2

Типичный класс, принимает аргументы в конструктор и есть какие-то методы, которые работают с этими данными. Как такое тестировать? Да не сложнее функций:

**class** **Test**(object):

"""

>>> a=Test(5)

>>> a.multiply\_by\_2()

10

"""

**def** **\_\_init\_\_**(self, number):

self.number = number

**def** **multiply\_by\_2**(self):

**return** self.number\*2

**if** \_\_name\_\_ == "\_\_main\_\_":

**import** doctest

doctest.testmod()

Что мы тут видим - в docstring такой же код из REPL как и для функций.

А вот пример выполнения теста:

> python example.py -v

Trying:

a=Test(5)

Expecting nothing

ok

Trying:

**a**.multiply\_by\_2()

Expecting:

10

ok

3 items had no tests:

\_\_main\_\_

\_\_main\_\_.Test.\_\_init\_\_

\_\_main\_\_.Test.multiply\_by\_2

1 items passed all tests:

2 tests **in** \_\_main\_\_.Test

2 tests **in** 4 items.

2 passed and 0 failed.

Test passed.

Доказал, что для простого класса и функций тестирование слабо отличается?

Много тестов - много проблем?

Очевидный недостаток doctest - когда тестов становится много, то неудобно писать в docstring. Рекомендуется выносить в отдельный файл.

Для такого тестирования в doctest есть функция:

**doctest**.testfile("**test**.txt")

В test.txt помещаете текст из docstring. Например так:

Тестирование функции **mult(a,b)**

>>> from test\_in\_other\_file **import** **mult**

>>> **mult(2,3)**

6

Выводы

По сравнению с классическими юнит-тестами, у доктестов есть как плюсы:

* простота написания тестов - можно скопировать прямо из REPL
* документация всегда соответствует коду

так и минусы:

* сложный код быстро становится не читаемым
* текстовый редактор не подсветит такой код,
* статический анализатор не найдет в нем ошибок
* подходит не для всех функций

Впрочем, ничто не мешает применять докстесты для мелких очевидных вещей (как в примере), и юнит-тесты для более сложных задач.