Занятие 4: Тестирование

Практикум на ЭВМ 2017/2018

Попов Артём Сергеевич

МГУ имени М. В. Ломоносова, факультет ВМК, кафедра ММП

Введение

В больших проектах без тестирования жить сложно:

- \+ Дополнительные проверки корректности кода
- \+ Быстрая проверка корректности при изменениях в коде
- – Требуется время на написание
- – Код для тестов может быть длиннее кода программы

Исключения

Исключения (exceptions) — проблемы, возникающие в ходе выполнения программы, приводящие к невозможности дальнейшей отработки программой её базового алгоритма.

Обработка исключений — механизм, предназначенный для описания реакции программы на исключения

В Python исключения являются объектами, через которые можно получить информацию об ошибке.

```
>>> 1 / 0
ZeroDivisionError: division by zero
>>> ZeroDivisionError.__doc__ # документация к исключению
'Second argument to a division or modulo operation was zero.'
>>> ZeroDivisionError.__mro__
(ZeroDivisionError, ArithmeticError, Exception,
BaseException, object)
```

Обработка исключений

Связка `try ... except` позволяет перехватывать исключения, возбужденные интерпретатором или программным кодом, и выполнять восстановительные операции:

```
>>> def one(x):
...     try:
...         print(x / x)
...     except ZeroDivisionError: # перехватывается конкретное
...         print('We did it!')          # исключение
...     except:                    # перехватываются все
...         print('Something is wrong') # исключения
...     # остальная часть программы
...
>>> one(5)
1
>>> one(0)
'We did it!'
>>> one('not a number')
'Something is wrong'
```

Обработка исключений

`raise` позволяет возбудить исключение программно:

```
>>> def one(x):
...     if type(x) != int:
...         raise TypeError('My message') # любое сообщение
...     return x / x
...
>>> one('string')
TypeError: My message
```

Можно создавать пользовательские исключения:

```
>>> class MyError(Exception):
...     pass
```

Юнит-тестирование

Общие принципы:

- Код программы разбит на независимые части
- Каждая часть тестируется отдельно и независимо

Признаки хорошего теста:

- Корректность
- Понятность
- Конкретность (проверяет что-то одно)
- Полезность

Функция для тестирования

Функция, которую мы будем тестировать:

```
>>> def factorial(x):
...     if x <= 0:
...         return 0
...     elif x == 1:
...         return 1
...     else:
...         return x * factorial(x - 1)
```

Самый простой вариант тестирования:

```
>>> print(factorial(3))
6
```

Тестирование вручную

Оператор assert возбуждает исключение, при невыполнении определенного условия:

```
>>> assert 1 == 2, 'Error' # второй аргумент - любое сообщение
AssertionError: Error
```

Напишем несколько тестов:

```
>>> def test_factorial(): # плохой
...     assert factorial(5) == (lambda n: [1, 0][n > 1] or
...                             fact(n - 1) * n)(5)
```

```
>>> def test_factorial(): # плохой
...     assert factorial(4) == 24, 'Positive numbers'
...     assert factorial(0) == 1, 'Zero'
...     try:
...         factorial(1000)
...     except RecursionError:
...         assert False, 'Recursion problem'
```

Тестирование вручную

```
>>> def test_factorial(): # плохой
...     assert factorial(5) != factorial(12)
```

```
>>> def test_factorial(): # плохой
...     assert factorial('string') != factorial(12)
```

```
>>> def test_factorial(): # хороший
...     assert factorial(0) == 1, 'Zero'
```

Тестирование вручную:

- \+ Быстро и легко писать
- – Надо запускать вручную

Модуль unittest

unittest — фреймворк для автоматизации тестов.
Тест — метод экземпляра наследника unittest.TestCase, начинающийся на test_.

```
>>> # модуль test_factorial.py
>>> import unittest
>>> from my_module import factorial
...
>>> class Test_factorial(unittest.TestCase):
...     def test_factorial_positive(self):
...         self.assertEqual(factorial(5), 120)
...
...     def test_factorial_zero(self):
...         self.assertEqual(factorial(0), 5) # специально!
...
>>> if __name__ == '__main__':
...     unittest.main() # запускает все тесты модуля
```

Результат работы unittest

```
python -m unittest test_factorial.py


.F
======================================================================
FAIL: test_factorial_zero (test_factorial.Test_factorial)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_factorial.py", line 10, in test_factorial_zero
    self.assertEqual(factorial(0), 5)
AssertionError: 1 != 5

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

Контекст тестов

В unittest можно запускать тесты с контекстом:

```
>>> # модуль test_factorial.py
>>> import unittest
>>> from my_module import factorial
...
>>> class Test_factorial(unittest.TestCase):
...     def setUp(self): # вызывается перед каждым тестом
...         self.base = open('file_with_examples')
...
...     def test_factorial_positive(self):
...         self.assertEqual(factorial(5), 120)
...
...     def tearDown(self): # вызывается после каждого теста
...         self.base.close()
...
>>> if __name__ == '__main__':
...     unittest.main()
```

Функции для проверки

Модуль unittest предоставляет множество функций для самых различных проверок, например:

- `assertIs(a, b)` — `a is b`
- `assertRaises(exc, func, *args, **kwds)` — `func(*args, **kwds)` порождает исключение exc

Дополнительные виды проверок можно получить из модуля `numpy.testing`:

- `assert_array_almost_equal(x, y, decimal)` — `x` и `y` совпадают с некоторой точностью

Кратко о unittest

+ Автоматически запускает тесты
+ Генерирует хорошие сообщения об ошибках
+ Много возможностей (см. документацию)
+ Есть в стандартной библиотеке
– Имеет нестандартный, непривычный интерфейс

Модуль py.test

py.test — альтернатива unittest для написания тестов.

У py.test минимальные требования к интерфейсу, тест это:

- функция с именем, начинающимся с test_
- то же, что и у unittest

```
>>> # модуль test_factorial.py
>>> import py.test
>>> from my_module import factorial
...
>>> def test_positive():
...     assert factorial(4) == 24
...
>>> def test_zero():
...     assert factorial(0) == 5 # специально!
```

Результат работы py.test

```
python3 -m pytest test_factorial.py
============================= test session starts ==============
platform linux -- Python 3.5.2, pytest-3.2.1, py-1.4.34
rootdir: /home/user/Programs, inifile:
collected 2 items

test_factorial.py .F

=================================== FAILURES ===================
_________________________________ test_zero _____________________

    def test_zero():
>       assert factorial(0) == 5
E       assert 1 == 5
E        +  where 1 = factorial(0)

test_factorial.py:19: AssertionError
====================== 1 failed, 1 passed in 0.02 seconds ======
```

Кратко о `py.test`

+ Автоматически запускает тесты
+ Генерирует хорошие сообщения об ошибках
+ Много возможностей (см. документацию)
+ Простой интерфейс
– Более магический чем `unittest` (но не в наших примерах)

doc.test

doctest позволяет писать тесты внутри документации.
doctest проводит сравнение (как строк!) записанного ответа и вывода интерпретатора на инструкцию:

```
import doctest
def factorial(x):
    """
    Documentation

    >>> factorial(5)
    120

    >>> factorial(0) # специально
    5
    """
    # код функции

doctest.testmod()
```

Результат работы doc.test

```
**********************************************************************
File "__main__", line 8, in __main__.factorial
Failed example:
    factorial(0) # специально
Expected:
    5
Got:
    1
**********************************************************************
1 items had failures:
   1 of   2 in __main__.factorial
***Test Failed*** 1 failures.
```

Кратко о `doc.test`

+ Автоматически запускает тесты
+ Генерирует хорошие сообщения об ошибках
+ В документации всегда актуальные примеры кода
- Результаты сравниваются только как строки
- Если в середине теста произошла ошибка, оставшиеся тесты не выполнятся

Заключение

В Python большое количество способов тестировать свой код:

- Ручное тестирование с помощью `assert`
- Прописывание простых тестов внутри документации с помощью `doc.test`
- Продвинуты библиотеки `unittest` и `py.test`

Писать тесты к своему коду полезно! Но всем нужна дополнительная мотивация...

За написание адекватных тестов к большим практическим заданиям будут добавляться бонусные баллы!