devpractice.ru

Python

unittest

Первое издание

Дорогие друзья! Мы дарим вам книгу “Python. unittest” абсолютно БЕСПЛАТНО! Если вы хотите **поддержать** коллектив авторов, то можете помочь проекту devpractice.ru любой суммой.

Yandex.Кошелек: 410011113064717
Сайт визитка для перевода: https://money.yandex.ru/to/410011113064717
Страница для поддержки проекта на нашем сайте http://devpractice.ru/donation/

Спасибо за помощь, это очень важно для нас!

Содержание

Глава 1. Введение

Автономное тестирование. Основные понятия

Трудно представить какой-то современный программный проект без тестирования. При этом тестирование осуществляется практических на всех этапах разработки продукта: начиная, непосредственно, с процесса создания функций, методов и классов и т.д., когда пишутся *unit*-тесты (а иногда и раньше, в случае, если используется *TDD*), и заканчивая функциональным и нагрузочным тестированием уже готового, развернутого продукта.

В рамках данной книги, мы остановимся только на автономном тестировании. В качестве определения данного понятия воспользуемся тем, что дает Рой Ошероув в своей книге “Искусство автономного тестирования с примерами на *C#*”: автономный тест – это автоматизированная часть кода, которая вызывает тестируемую единицу работы и затем проверяет некоторые предположения о единственном конечном результате этой единицы. В качестве тестируемый единицы, в данном случае, может выступать как отдельный метода (функция), так и совокупность классов (или функций). Идея автономной единицы в том, что она представляет собой некоторую логически законченную сущность вашей программы. Автономное тестирование еще называют модульным или *unit*-тестированием (*unit-testing*). Здесь и далее под словом тестирование будет пониматься именно автономное тестирование.

Важной характеристикой *unit*-теста является его повторяемость, т.е. результат его работы не зависит от окружения (внешнего мира), если же приходится обращаться к внешнему миру в процессе выполнения теста, то необходимо предусмотреть возможность подмены “мира” какой-то статичной сущностью.

Unit-тесты могут быть написаны собственноручно, без использования сторонних библиотек, а можно использовать специализированные *framework’u*. На сегодняшний день практически всегда используется второй вариант.

Framework’u для проведения автономного тестирования в *Python*

В мире *Python* существуют три *framework’a*, которые получили наибольшее распространение:

- *unittest*
- *nose*
- *pytest*

unittest

unittest – это *framework* для тестирования, входящий в стандартную библиотеку языка *Python*. Его архитектура выполнена в стиле *xUnit*. *xUnit* представляет собой семейство *framework'ов* для тестирования в разных языках программирования, в *Java* – это *JUnit*, *C# – NUnit* и т.д. Если вы уже сталкивались с данным каркасов в других языках, то это упростит понимание *unittest*. Т.к. данная книга посвещена *unittest*, то мы не будет сейчас подробно на нем останавливаться.

nose

Девизом *nose* является фраза "*nose extends unittest to make testing easier*", что можно перевести как "*nose* расширяет *unittest*, делая тестирование проще". *nose* идеален, когда нужно сделать тесты "по-быстрому", без предварительного планирования и выстраивания архитектуры приложения с тестами. Функционал *nose* можно расширять и настраивать с помощью плагинов.

pytest

pytest довольно мощный инструмент для тестирования, и многие разработчики оставляют свой выбор именно на нем. *pytest* по "духу" ближе к языку *Python* нежели *unittest*. Как было сказано выше, *unittest* в своей базе – *xUnit*, что накладывает определенные обязательства при разработке тестов (создание классов-наследников от *unittest.TestCase*, выполнение определенной процедуры запуска тестов и т.п.). При разработке на *pytest* ничего этого делать не нужно, вы просто пишете функции, которые должны начинаться с "*test_*" и используете *assert'ы*, встроенные в *Python* (*unittest* использует свои). У *pytest* есть ещё много интересных и полезных особенностей, но это тема для отдельной книги.

Пример тестирования приложения без *framework'a*

Рассмотрим простейший модуль *Python*, который содержит ряд функций, и разберем пример того, как можно было бы его протестировать без использования *framework'a*. Наш модуль будет представлять собой библиотеку, содержащую функции для выполнения простых арифметический действий.

Модуль calc.py

```
def add(a, b):
    return a + b

def sub(a, b):
    return a-b

def mul(a, b):
    return a * b

def div(a, b):
    return a / b
```

Для того, чтобы протестировать эту библиотеку, мы можем создать отдельный файл с названием *test_calc.py* и поместить туда функции, которые проверяют корректность работы функций из *calc.py*.

Модуль test_calc.py

```
import calc

def test_add():
    if calc.add(1, 2) == 3:
        print("Test add(a, b) is OK")
    else:
        print("Test add(a, b) is Fail")

def test_sub():
    if calc.sub(4, 2) == 2:
        print("Test sub(a, b) is OK")
    else:
        print("Test sub(a, b) is Fail")

def test_mul():
    if calc.mul(2, 5) == 10:
        print("Test mul(a, b) is OK")
    else:
        print("Test mul(a, b) is Fail")

def test_div():
    if calc.div(8, 4) == 2:
        print("Test div(a, b) is OK")
    else:
        print("Test div(a, b) is Fail")
test_add()
test_sub()
```

```
test_mul()
test_div()
```

Запустим *test_calc.py*.

> python test_calc.py

В результате, в окне консоли, будет напечатано следующее:

```
Test add(a, b) is OK
Test sub(a, b) is OK
Test mul(a, b) is OK
Test div(a, b) is OK
```

Это были четыре теста, которые проверяют работоспособность функций в простейшем случае. При написании тестов, как обычных программ, возникает ряд неудобств, в первую очередь связанных с унификацией выходной информации о пройденных и непройденных тестах, сами тесты получаются довольно громоздкими, также необходимо продумывать архитектуру тестирующего приложения и т.д. В дополнение к этому можно отметить отсутствие гибких инструментов для запуска требуемых только на данном этапе тестов, пропуска тестов по условию (например для разрабатываемой библиотеки, начиная с определённой версии, не выполнять конкретные тесты) и т.п. Все это приводит к мысли о том, что нужен какой-то *framework*, который возьмет на себя обязанности по поддержанию инфраструктуры проекта с тестами.

Пример тестирования приложения с использованием *unittest*

Теперь посмотрим как можно было бы протестировать набор функций из *calc.py* с помощью *unittest*.

Для этого сделаем следующие действия:

1. Создадим файл с именем *utest_calc.py.*

2. Добавим в него следующий код:

```
import unittest
import calc

class CalcTest(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

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

3. Запустим файл *utest_calc.py.*

> python -m unittest utest_calc.py

Такой формат запуска предполагает вывод минимальной информации. В данном случае все тесты успешно завершились.

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s
OK
```

4. Запуск можно сделать с запросом расширенной информации по пройденным тестам, для этого необходимо добавить ключ *-v*:

> python -m unittest -v utest_calc.py

В этом случае результат будет таким:

```
test_add (test_calc_v2.CalcTest) ... ok
test_div (test_calc_v2.CalcTest) ... ok
test_mul (test_calc_v2.CalcTest) ... ok
test_sub (test_calc_v2.CalcTest) ... ok
----------------------------------------------------------------------
Ran 4 tests in 0.002s
OK
```

На этом простом примере не видно всех преимуществ, которые дает *unittest*, по сравнению с вариантом без него. Для кого-то даже покажется лишним создание отдельных классов и запуск модулей с дополнительными ключами, но в более сложном случае, преимущества использования *framework'a* несомненны. В следующих главах мы постараемся последовательно и подробно раскрыть вопросы написания автономных тестов с использованием *unittest* в *Python*.

Глава 2. Написание тестов (класс *TestCase*)

В этой главе сосредоточимся на общем обзоре основных структурных элементов *unittest* (*test case, test suite, test runner*), рассмотрим способы запуска тестов и подробно остановимся на классе *TestCase*.

Основные структурные элементы *unittest*

unittest – это *framework* для тестирования в *Python*, который позволяет разрабатывать автономные тесты, собирать тесты в коллекции, обеспечивает независимость тестов от *framework'a* отчетов и т.д. Основными структурными элемента каркаса *unittest* являются (https://goo.gl/N1q41S):

Test fixture

Test fixture – обеспечивает подготовку окружения для выполнения тестов, а также организацию мероприятий по их корректному завершению (например очистка ресурсов). Подготовка окружения может включать в себя создание баз данных, запуск необходим серверов и т.п.

Test case

Test case – это элементарная единица тестирования, в рамках которой проверяется работа компонента тестируемой программы (метод, класс, поведение и т.п.). Для реализации этой сущности используется класс *TestCase*.

Test suite

Test suite – это коллекция тестов, которая может в себя включать как отдельные *test case'ы* так и целые коллекции (т.е. можно создавать коллекции коллекций). Коллекции используются с целью объединения тестов для совместного запуска.

Test runner

Test runner – это компонент, который оркестрирует (координирует взаимодействие) запуск тестов и предоставляет пользователю результат их выполнения. *Test runner* может иметь графический интерфейс, текстовый интерфейс или возвращать какое-то заранее заданное значение, которое будет описывать результат прохождения тестов.

Вся работа по написанию тестов заключается в том, что мы разрабатываем отдельные тесты в рамках *test case'ов*, собираем их в модули и запускаем, если нужно объединить несколько *test case'ов* для их совместного запуска, они помещаются в *test suite'ы*, которые помимо *test case'ов* могут содержать другие *test suite'ы*.

Запуск тестов

Запуск тестов можно сделать как из командной строки, так и с помощью графического интерфейса пользователя (*GUI*), рассмотрим каждый из этих способов более подробно. В качестве примера приложения, будет выступать *utest_calc.py* из предыдущей главы.

Интерфейс командной строки (*CLI*)

CLI позволяет запускать группы тесты из модуля или класса, а также обеспечивает доступ к каждому тесту по отдельности.

Запуск всех тестов в модуле *utest_calc.py*.

> python -m unittest test_calc.py

Запуск тестов из класса *CalcTest*.

> python -m unittest utest_calc.CalcTest

Запуск теста *test_sub()*.

> python -m unittest utest_calc.CalcTest.test_sub

Как уже было сказано в первой главе, для вывода подробной информации необходимо добавить ключ *-v*.

> python -m unittest -v utest_calc.py

Если осуществить запуск без указания модуля с тестами, то будет запущен *Test Discovery*, который проведет определенную работу по выполнению тестов.

> python -m unittest

Справку по ключам запуска и информацию о *Test Discovery* можно получить из документации (https://goo.gl/23YfJ7).

Графический интерфейс пользователя (*GUI*)

Для запуска и анализа результатов работы тестов можно использовать *GUI*. Список инструментов доступен на wiki (https://goo.gl/KUiSk6), но он далеко не полный.

Для примера, рассмотрим работу с *Cricket* (https://github.com/pybee/cricket). Для установки *Cricket* можно воспользоваться менеджером *pip*:

> pip install cricket

После этого на ваш компьютер будет установлен *cricket-unittest*.

Для запуска тестов в данном приложении, перейдите в каталог с вашим тестирующим кодом и в командной строке запустите *cricket-unittest*, для этого просто наберите название программы и нажмите *Enter*.

> cricket-unittest

Приложение, при запуске, автоматически загрузит тесты (см. рисунок 1).

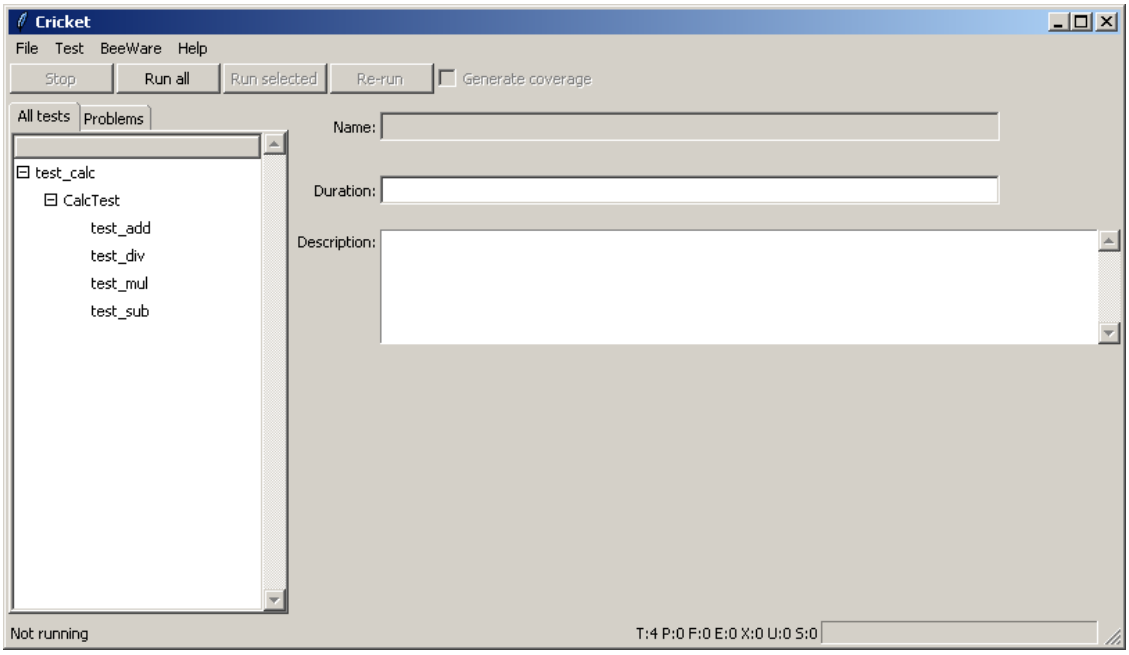

Рисунок 1 - Окно программы *Cricket* с загруженными тестами

Для запуска тестов нажмите “*Run all*”. Как видно, все тесты завершились удачно – они окрасились в зеленый цвет (см. рисунок 2).

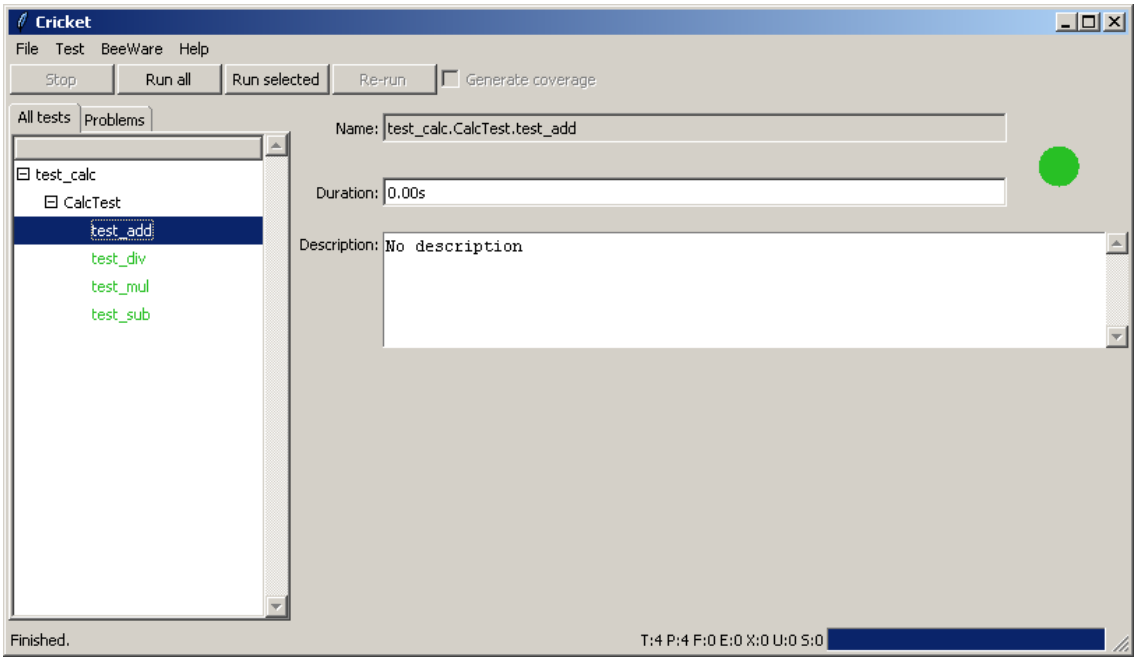

Рисунок 2 - Результат выполнения тестов в *Cricket*

Работа с *TestCase*

Как уже было сказано – основным строительным элементом при написании тестов с использованием *unittest* является *TestCase*. Он представляет собой базовый (родительский) класс для всех остальных классов, методы которых будут тестировать автономные единицы исходной программы. Вот содержимое класса *CalcTest* из предыдущей главы (модуль *utest_calc.py*).

```
import unittest
import calc
class CalcTests(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)
if __name__ == '__main__':
    unittest.main()
```

Для того, чтобы у нас появилась возможность использовать компоненты *unittest* (в том числе и *TestCase*), в самом начале программы нужно импортировать модуль *unittest* стандартным образом.

При выборе имени класса наследника от *TestCase* можете руководствоваться следующим правилом: *[ИмяТестируемойСущности]Tests*. *[ИмяТестируемойСущности]* – это некоторая логическая единица, тесты для которой нужно написать. В нашем случае – это калькулятор, поэтому мы выбрали имя *CalcTests*. Если бы у нашего калькулятора был большой набор поддерживаемых функций, то тестирование простых функций (сложение, вычитание, умножение и деление) можно было бы вынести в отдельный класс и назвать его например так: *CalcSimpleActionsTests*. При написании программ на Python старайтесь придерживаться *PEP 8 — Style Guide for Python Code* – это рекомендации по стилевому оформлению кода.

Для того, чтобы метод класса выполнялся как тест, необходимо, чтобы он начинался со слова ***test***. Несмотря на то, что методы *framework'a unittest* написаны не в соответствии с *PEP 8* (ввиду того, что идейно он наследник *xUnit*), мы все же рекомендуем следовать правилам стиля для *Python* везде, где это возможно. Поэтому имена тестов будем начинать с префикса *test_*. Далее, под словом тест будем понимать метод класса-наследника от *TestCase*, который начинается с префикса *test_*.

Все методы класса *TestCase* можно разделить на три группы:

- методы, используемые при запуске тестов;
- методы, используемые при непосредственном написании тестов (проверка условий, сообщение об ошибках);
- методы, позволяющие собирать информацию о самом тесте.

Рассмотрим методы этих групп более подробно. Остановимся только на тех методах, которые могут быть полезны в первую очередь, при разработке тестов. За более подробной информацией можете обратиться к официальной документации (https://goo.gl/25obHZ).

Методы, используемые при запуске тестов

К этим методам относятся:

setUp()

Метод вызывается перед запуском теста. Как правило, используется для подготовки окружения для теста.

tearDown()

Метод вызывается после завершения работы теста. Используется для “приборки” за тестом.

Заметим, что методы *setUp()* и *tearDown()* вызываются для всех тестов в рамках класса, в котором они переопределены. По умолчанию, эти методы ничего не делают. Если их добавить в *utest_calc.py*, то перед/после тестов *test_add(), test_sub(), test_mul(), test_div()* будут выполнены *setUp()*/*tearDown()* (см. рисунок 3).

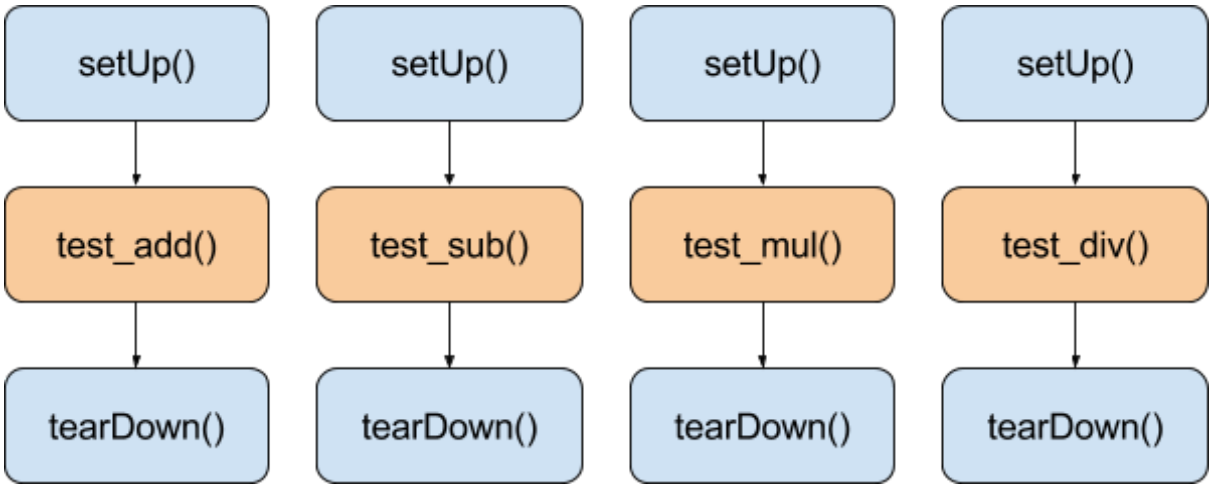

Рисунок 3 - Выполнение методов *setUp()* и *tearDown()* для тестов из *utest_calc.py*

setUpClass()

Метод действует на уровне класса, т.е. выполняется перед запуском тестов класса. При этом синтаксис требует наличие декоратора *@classmethod*.

```
@classmethod
def setUpClass(cls):
    ...
```

tearDownClass()

Запускается после выполнения всех тестов класса, требует наличия декоратора *@classmethod*.

```
@classmethod
def tearDownClass(cls):
    ...
```

skipTest(reason)

Данный метод может быть использован для пропуска теста, если это необходимо.

Методы, используемые при непосредственном написании тестов

TestCase класс предоставляет набор *assert*-методов для проверки и генерации ошибок.

Таблица 1 - Методы для проверки условий с генерацией ошибок

Метод	**Проверяемое условие**
assertEqual(a, b)	*a == b*
assertNotEqual(a, b)	*a != b*
assertTrue(x)	*bool(x) is True*
assertFalse(x)	*bool(x) is False*
assertIs(a, b)	*a is b*
assertIsNot(a, b)	*a is not b*
assertIsNone(x)	*x is None*
assertIsNotNone(x)	*x is not None*
assertIn(a, b)	*a in b*
assertNotIn(a, b)	*a not in b*
assertIsInstance(a, b)	*isinstance(a, b)*
assertNotIsInstance(a, b)	*not isinstance(a, b)*

Таблица 2 - *Assert'ы* для контроля выбрасываемых исключений и *warning'ов*

Метод	**Проверяемое условие**
*assertRaises(exc, fun, *args, **kwds)*	Функция *fun(*args, **kwds)* вызывает исключение *exc*
*assertRaisesRegex(exc, r, fun, *args, **kwds)*	Функция *fun(*args, **kwds)* вызывает исключение *exc*, сообщение которого совпадает с регулярным выражением *r*
*assertWarns(warn, fun, *args, **kwds)*	Функция *fun(*args, **kwds)* выдает сообщение *warn*
*assertWarnsRegex(warn, r, fun, *args, **kwds)*	Функция *fun(*args, **kwds)* выдает сообщение *warn* и оно совпадает с регулярным выражением *r*

Таблица 3 - *Assert'ы* для проверки различных ситуаций

Метод	**Проверяемое условие**
assertAlmostEqual(a, b)	*round(a-b, 7) == 0*
assertNotAlmostEqual(a, b)	*round(a-b, 7) != 0*
assertGreater(a, b)	*a > b*
assertGreaterEqual(a, b)	*a >= b*
assertLess(a, b)	*a < b*
assertLessEqual(a, b)	*a <= b*
assertRegex(s, r)	*r.search(s)*
assertNotRegex(s, r)	*not r.search(s)*
assertCountEqual(a, b)	*a* и *b* содержат одинаковые элементы (порядок неважен)

Таблица 4 - Типо-зависимые *assert'ы*, которые используются при вызове *assertEqual()*

Метод	**Проверяемое условие**
assertMultiLineEqual(a, b)	строки (*strings*)
assertSequenceEqual(a, b)	последовательности (*sequences*)

assertListEqual(a, b)	списки (*lists*)
assertTupleEqual(a, b)	кортежи (*tuplse*)
assertSetEqual(a, b)	множества или неизменяемые множества (*frozensets*)
assertDictEqual(a, b)	словари (*dicts*)

Дополнительно хотелось бы отметить метод *fail()*.

fail(msg=None)

Этот метод сигнализирует о том, что произошла ошибка в тесте.

Методы, позволяющие собирать информацию о самом тесте

countTestCases()

Возвращает количество тестов в объекте класса-наследника от *TestCase*.

id()

Возвращает строковый идентификатор теста. Как правило это полное имя метода, включающее имя модуля и имя класса.

shortDescription()

Возвращает описание теста, которое представляет собой первую строку *docstring'a* метода, если его нет, то возвращает *None*.

Расширим код нашего тестового проекта *utest_calc.py*, так чтобы показать некоторые из возможностей, которые предоставляет класс *TestCase*.

```
import unittest
import calc
class CalcTest(unittest.TestCase):
    """Calc tests"""
    @classmethod
    def setUpClass(cls):
        """Set up for class"""
        print("setUpClass")
        print("==========")

    @classmethod
    def tearDownClass(cls):
        """Tear down for class"""
        print("==========")
        print("tearDownClass")

    def setUp(self):
        """Set up for test"""
        print("Set up for [" + self.shortDescription() + "]")

    def tearDown(self):
        """Tear down for test"""
        print("Tear down for [" + self.shortDescription() + "]")
        print("")

    def test_add(self):
        """Add operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        """Sub operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        """Mul operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        """Div operation test"""
        print("id: " + self.id())
        self.assertEqual(calc.div(8, 4), 2)
if __name__ == '__main__':
    unittest.main()
```

Запустив это модуль в командной строке:
> python -m unittest -v utest_calc.py

Получим следующий результат:

```
setUpClass
==========
test_add (simple_ex.CalcTest)
Add operation test ... Set up for [Add operation test]
id: simple_ex.CalcTest.test_add
Tear down for [Add operation test]
ok

test_div (simple_ex.CalcTest)
Div operation test ... Set up for [Div operation test]
id: simple_ex.CalcTest.test_div
Tear down for [Div operation test]
ok

test_mul (simple_ex.CalcTest)
Mul operation test ... Set up for [Mul operation test]
id: simple_ex.CalcTest.test_mul
Tear down for [Mul operation test]
ok

test_sub (simple_ex.CalcTest)
Sub operation test ... Set up for [Sub operation test]
id: simple_ex.CalcTest.test_sub
Tear down for [Sub operation test]
ok
==========
tearDownClass
----------------------------------------------------------------------
Ran 4 tests in 0.016s
OK
```

Как видно из примера, вначале был запущен метод *setUpClass()*, потом последовательно (в алфавитном порядке) были выполнены тесты, перед запуском каждого теста выполнялся метод *setUp()*, по окончании – *tearDown()*. Каждый метод содержит *docstring* в виде комментария в первой строке. Для доступа к этому описанию использовался метод *shortDescription()*. В теле теста присутствует строка, печатающая идентификатор, получаемый с помощью функции *id()*.

Глава 3. Организация тестов (класс *TestSuite*). Загрузка и запуск тестов

Третья глава посвящена *TestSuite* – второй важной составляющей *unittest*, а также загрузке и запуску тестов (классы *TestLoader, TestResult, TextTestRunner*).

Для более полного изучения возможностей рассматриваемых классов рекомендуем обратиться к официальной документации (https://goo.gl/MLMT6m).

Класс *TestSuite*

Класс *TestSuite* используется для объединения тестов в группы, которые могут включать в себя как отдельные тесты так и заранее созданные группы. Помимо этого, *TestSuite* предоставляет интерфейс, позволяющий *TestRunner'y*, запускать тесты. Разберем более подробно методы класса *TestSuite*.

addTest(test)

Добавляет *TestCase* или *TestSuite* в группу.

addTests(tests)

Добавляет все *TestCase* и *TestSuite* объекты в группу, итеративно проходя по элементам переменной *tests*.

run(result)

Запускает тесты из данной группы.

countTestCases()

Возвращает количество тестов в данной группе (включает в себя как отдельные тесты, так и подгруппы).

Рассмотрим пример использования *TestSuite*.

В качестве кода, который нужно протестировать, возьмем уже знакомый нам модуль *calc.py*.

```
def add(a, b):
    return a + b

def sub(a, b):
    return a-b

def mul(a, b):
    return a * b

def div(a, b):
    return a / b
```

За основу модуля с тестами примем тот, что приведен в конце первой главы (*calc_tests.py*).

```
import unittest
import calc

class CalcTest(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)
```

Для запуска тестов дополнительно создадим модуль *test_runner.py* и добавим в него следующий код.

```
import unittest
import calc_tests

calcTestSuite = unittest.TestSuite()
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcTest))

runner = unittest.TextTestRunner(verbosity=2)
runner.run(calcTestSuite)
```

Все модули должны находиться в одном каталоге. Для запуска тестов используйте команду:

>python test_runner.py

В примере мы использовали класс *TextTestRunner*, о нем будет рассказано чуть позже.

Расширим функционал модуля *calc.py*, для этого добавим в него пару методов: первый будет вычислять квадратный корень, второй – возводить число в определенную степень.

Модуль calc.py

```
def add(a, b):
    return a + b

def sub(a, b):
    return a-b

def mul(a, b):
    return a * b

def div(a, b):
    return a / b

def sqrt(a):
    return a**0.5

def pow(a, b):
    return a**b
```

Добавим тесты для новых функций: создадим новый класс с именем *CalcExTests* (расширенные функции калькулятора) с тестами для *sqrt()* и *pow()*, а класс *CalcTest* переименуем в *CalcBasicTests* (базовые функции калькулятора).

Модуль calc_tests.py

```
import unittest
import calc

class CalcBasicTests(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

class CalcExTests(unittest.TestCase):
    def test_sqrt(self):
        self.assertEqual(calc.sqrt(4), 2)

    def test_pow(self):
        self.assertEqual(calc.pow(3, 3), 27)
```

Модуль test_runner.py

```
import unittest
import calc_tests

calcTestSuite = unittest.TestSuite()
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcBasicTests))
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcExTests))
print("count of tests: " + str(calcTestSuite.countTestCases()) + "\n")

runner = unittest.TextTestRunner(verbosity=2)
runner.run(calcTestSuite)
```

Запустив *test_runner.py* получим следующий результат.

```
count of tests: 6
test_add (calc_tests.CalcBasicTests) ... ok
test_div (calc_tests.CalcBasicTests) ... ok
test_mul (calc_tests.CalcBasicTests) ... ok
test_sub (calc_tests.CalcBasicTests) ... ok
test_pow (calc_tests.CalcExTests) ... ok
test_sqrt (calc_tests.CalcExTests) ... ok
----------------------------------------------------------------------
Ran 6 tests in 0.000s
OK
```

Как видно из примера: было запущено шесть тестов, четыре из класса *CalcBasicTests* и два из *CalcExTests*, все тесты завершились удачно. Количество тестов в группе указано в самой первой строке вывода: *count of tests: 6*.

Загрузка и запуск тестов

Рассмотрим более подробно вопросы загрузки и запуска тестов.

Класс *TestLoader*

Начнем с класса *TestLoader*. Этот класс используется для создания групп из классов и модулей. Среди методов *TestLoader* можно выделить: *loadTestsFromTestCase(testCaseClass)*, возвращающий группу со всеми тестами из класса *testCaseClass*. Напоминаем, что под тестом понимается метод, начинающийся со слова “*test*”. Используя этот *loadTestsFromTestCase*, можно создать список групп тестов, где каждая группа создается на базе классов-наследников от *TestCase*, объединенных предварительно в список. Для демонстрации данного подхода модифицируем *test_runner.py* (написано по материалам *dr.dobb’s (https://goo.gl/juxvAa)*).

Модуль test_runner.py

```
import unittest
import calc_tests

testCases = []
testCases.append(calc_tests.CalcBasicTests)
testCases.append(calc_tests.CalcExTests)

testLoad = unittest.TestLoader()

suites = []
for tc in testCases:
    suites.append(testLoad.loadTestsFromTestCase(tc))

res_suite = unittest.TestSuite(suites )

runner = unittest.TextTestRunner(verbosity=2)
runner.run(res_suite)
```

Рассмотрим ещё несколько методов из *TestLoader*.

loadTestsFromModule(module, pattern=None)

Загружает все тесты из модуля *module*. Если модуль поддерживает *load_tests* протокол, то будет вызвана соответствующая функция модуля и ей будет передан в качестве аргумента (третьим по счету) параметр *pattern*.

loadTestsFromName(name, module=None)

Загружает тесты в соответствии с параметром *name*. Параметр name – это имя, разделенное точками. С помощью этого имени указывается уровень, начиная с которого будут добавляться тесты.

getTestCaseNames(testCaseClass)

Возвращает список имен методов-тестов из класса *testCaseClass*.

Приведем примеры того, как можно использовать данные методы. Для демонстрации *loadTestsFromModule* изменим модуль *test_runner.py*.

Модуль test_runner.py

```
import unittest
import calc_tests

testLoad = unittest.TestLoader()
suites = testLoad.loadTestsFromModule(calc_tests)

runner = unittest.TextTestRunner(verbosity=2)
runner.run(suites)
```

Запустим модуль *test_runner.py.*
>python test_runner.py

Результатом выполнения будет подробный отчет о прохождении шести тестов:

```
test_add (calc_tests.CalcBasicTests) ... ok
test_div (calc_tests.CalcBasicTests) ... ok
test_mul (calc_tests.CalcBasicTests) ... ok
test_sub (calc_tests.CalcBasicTests) ... ok
test_pow (calc_tests.CalcExTests) ... ok
test_sqrt (calc_tests.CalcExTests) ... ok
----------------------------------------------------------------------
Ran 6 tests in 0.016s
OK
```

Если в модуле *test_runner.py* заменить строку

suites = testLoad.loadTestsFromModule(calc_tests)

на

suites = testLoad.loadTestsFromName("calc_tests.CalcBasicTests")

то будут выполнены только тесты из класса *CalcBasicTests*.

```
test_add (calc_tests.CalcBasicTests) ... ok
test_div (calc_tests.CalcBasicTests) ... ok
test_mul (calc_tests.CalcBasicTests) ... ok
test_sub (calc_tests.CalcBasicTests) ... ok
----------------------------------------------------------------------
Ran 4 tests in 0.002s
OK
```

Класс *TestResult*

Класс *TestResult* используется для сбора информации о результатах прохождения тестов. Подробную информацию по атрибутам и классам этого метода можно найти в официальной документации *(https://goo.gl/NmEer9).*

Для демонстрации возможностей класса *TestResult* модифицируем модуль *test_runner.py*:

```
import unittest
import calc_tests

testLoad = unittest.TestLoader()
suites = testLoad.loadTestsFromModule(calc_tests)

testResult = unittest.TestResult()

runner = unittest.TextTestRunner(verbosity=1)
testResult = runner.run(suites)
print("errors")
print(len(testResult.errors))
print("failures")
print(len(testResult.failures))
print("skipped")
print(len(testResult.skipped))
print("testsRun")
print(testResult.testsRun)
```

Запустив его, получим следующий результат:

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.001s
OK
errors
0
failures
0
skipped
0
testsRun
6
```

Класс *TextTestRunner*

Объекты класса *TextTestRunner* используются для запуска тестов. Среди параметров, которые передаются конструктору класса, можно выделить *verbosity*, по умолчанию он равен 1, если создать объект с *verbosity=2*, то будем получать расширенную информацию о результатах прохождения тестов. Для запуска тестов используется метод *run()*, которому в качестве аргумента передается класс-наследник от *TestCase* или группа (*TestSuite*).

В наших примерах *TextTestRunner* используется в модуле *test_runner.py* в строчках:

runner = unittest.TextTestRunner(verbosity=2)

testResult = runner.run(suites)

В первой строке создается объект класса *TextTestRunner* с *verbosity=2*, а во второй строке запускаются тесты из группы *suites*, результат тестирования попадает в объект *testResult*, атрибуты которого можно анализировать в дальнейшем.

Глава 4. Пропуск тестов

В рамках четвертой главы изучим вопрос пропуска тестов. Будет рассмотрен условный и безусловный пропуск тестов, а также пропуск всех тестов внутри класса.

Подготовка

В качестве тестируемого модуля будем использовать расширенный модуль *calc.py* из главы 3, модуль с тестами - *calc_tests.py* и модуль для запуска тестов - *test_runner.py*.

Модуль calc.py

```
def add(a, b):
    return a + b

def sub(a, b):
    return a-b

def mul(a, b):
    return a * b

def div(a, b):
    return a / b

def sqrt(a):
    return a**0.5

def pow(a, b):
    return a**b
```

Модуль calc_tests.py

```
import unittest
import calc

class CalcBasicTests(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

class CalcExTests(unittest.TestCase):
    def test_sqrt(self):
        self.assertEqual(calc.sqrt(4), 2)

    def test_pow(self):
        self.assertEqual(calc.pow(3, 3), 27)
```

Модуль test_runner.py

```
import unittest
import calc_tests

calcTestSuite = unittest.TestSuite()
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcBasicTests))
calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcExTests))

runner = unittest.TextTestRunner(verbosity=2)
runner.run(calcTestSuite)
```

Пропуск отдельных тестов в классе

Безусловный пропуск тестов

Для начала запустим модуль *test_runner.py* (о запуске тестов можно прочитать в третьей главе). Получим следующий результат:

```
test_add (calc_tests.CalcBasicTests) ... ok
test_div (calc_tests.CalcBasicTests) ... ok
test_mul (calc_tests.CalcBasicTests) ... ok
test_sub (calc_tests.CalcBasicTests) ... ok
test_pow (calc_tests.CalcExTests) ... ok
test_sqrt (calc_tests.CalcExTests) ... ok
----------------------------------------------------------------------
Ran 6 tests in 0.002s
OK
```

Исключим тест *test_add* из списка тестов. При попытке решить такую задачу, первое, что может прийти на ум – это удалить либо закомментировать данный тест. Но *unittest* предоставляет нам инструменты для удобного управление процессом пропуска тестов. Это может быть ещё полезно в том плане, что информацию о пропущенных тестах (их количестве) можно дополнительно получить через специальный *API*, предоставляемый классом *TestResult*. Для пропуска теста воспользуемся декоратором *@unittest.skip(reason)*, который пишется перед тестом

Модифицируем класс *CalcBasicTests* из модуля *calc_tests.py*.

```
class CalcBasicTests(unittest.TestCase):
    @unittest.skip("Temporary skip test_add")
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)
```

И снова запустим *test_runner.py*.

```
test_add (calc_tests.CalcBasicTests) ... skipped 'Temporarily skipped'
test_div (calc_tests.CalcBasicTests) ... ok
test_mul (calc_tests.CalcBasicTests) ... ok
test_sub (calc_tests.CalcBasicTests) ... ok
test_pow (calc_tests.CalcExTests) ... ok
test_sqrt (calc_tests.CalcExTests) ... ok
----------------------------------------------------------------------
Ran 6 tests in 0.003s
OK (skipped=1)
```

Как видно из примера, был пропущен один тест – *test_add*.

Условный пропуск тестов

Для условного пропуска тестов применяются следующие декораторы:
@unittest.skipIf(condition, reason)
Тест будет пропущен, если условие (*condition*) истинно.

@unittest.skipUnless(condition, reason)
Тест будет пропущен если, условие (*condition*) не истинно.

Условный пропуск тестов можно использовать в ситуациях, когда те или иные тесты зависят от версии программы, например: в новой версии уже не поддерживается часть методов; или тесты могут быть платформозависимые, например: ряд тестов могут выполняться только под операционной системой *MS Windows*. Условие записывается в параметр *condition*, текстовое описание – в *reason*.

Пропуск классов

Для пропуска классов используется декоратор *@unittest.skip(reason),* который записывается перед объявлением класса. В результате все тесты из данного класса не будут выполнены. В рамках нашего примера с математическими действиями, для исключения из процесса тестирования методов *sqrt* и *pow* поместим декоратор *skip* перед объявлением класса *CalcExTests*.

Модуль calc_tests.py

```
import unittest
import calc

class CalcBasicTests(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(1, 2), 3)

    def test_sub(self):
        self.assertEqual(calc.sub(4, 2), 2)

    def test_mul(self):
        self.assertEqual(calc.mul(2, 5), 10)

    def test_div(self):
        self.assertEqual(calc.div(8, 4), 2)

@unittest.skip("Skip CalcExTests")
class CalcExTests(unittest.TestCase):
    def test_sqrt(self):
        self.assertEqual(calc.sqrt(4), 2)

    def test_pow(self):
        self.assertEqual(calc.pow(3, 3), 27)
```

Результат будет следующим:

```
test_add (calc_tests.CalcBasicTests) ... ok
test_div (calc_tests.CalcBasicTests) ... ok
test_mul (calc_tests.CalcBasicTests) ... ok
test_sub (calc_tests.CalcBasicTests) ... ok
test_pow (calc_tests.CalcExTests) ... skipped 'Skip CalcExTests'
test_sqrt (calc_tests.CalcExTests) ... skipped 'Skip CalcExTests'
----------------------------------------------------------------------
Ran 6 tests in 0.001s
OK (skipped=2)
```