Python必会的单元测试框架 —— unittest

2016年10月27日 12:52:37 标签：python / 单元测试 / 框架 / 自动化测试 / unittest 20047

25

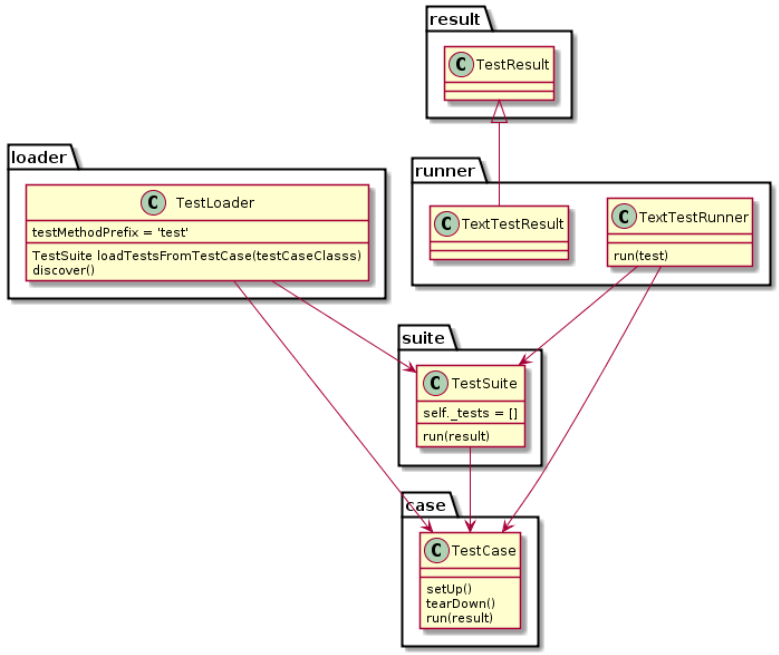
用Python搭建自动化测试框架，我们需要组织用例以及测试执行，这里博主推荐Python的标准库——unittest。

unittest是xUnit系列框架中的一员，如果你了解xUnit的其他成员，那你用unittest来应该是很轻松的，它们的工作方式都差不多。

unittest核心工作原理

unittest中最核心的四个概念是：**test case, test suite, test runner, test fixture**。

下面我们分别来解释这四个概念的意思，先来看一张unittest的静态类图（下面的类图以及解释均来源于网络，[文链接]）：



- 一个TestCase的实例就是一个测试用例。什么是测试用例呢？就是一个完整的测试流程，包括测试前准备环境的搭建(setUp)，执行测试代码(run)，以及测试后环境的还原(tearDown)。元测试(unit test)的本质也就在这里，一个测试用例是一个完整的测试单元，通过运行这个测试单元，可以对某一个问题进行验证。
- 而多个测试用例集合在一起，就是TestSuite，而且TestSuite也可以嵌套TestSuite。
- TestLoader是用来加载TestCase到TestSuite中的，其中有几个loadTestsFrom__()方法，就是从各个地方寻找TestCase，创建它们的实例，然后add到TestSuite中，再返回一个TestSuite实例。
- TextTestRunner是来执行测试用例的，其中的run(test)会执行TestSuite/TestCase中的run(result)方法。
  测试的结果会保存到TextTestResult实例中，包括运行了多少测试用例，成功了多少，失败了多少等信息。
- 而对一个测试用例环境的搭建和销毁，是一个fixture。

一个class继承了unittest.TestCase，便是一个测试用例，但如果其中有多个以 `test` 开头的方法，那么每有一个这样的方法，在load的时候便会生成一个TestCase实例，如：一个class中有四个test_xxx方法，最后在load到suite中时也有四个测试用例。

到这里整个流程就清楚了：

写好TestCase，然后由TestLoader加载TestCase到TestSuite，然后由TextTestRunner来运行TestSuite，运行的结果保存在TextTestResult中，我们通过命令行或者unittest.main()执行时，main会调用TextTestRunner中的run来执行，或者我们可以直接通过TextTestRunner来执行用例。这里加个说明，在Runner执行时，默认将执行结果输出到控制台，我们可以设置其输出到文件，在文件中查看结果（你可能听说过HTMLTestRunner，是的，

通过它可以将结果输出到HTML中，生成漂亮的报告，它跟TextTestRunner是一样的，从名字就能看出来，这个我们后面再说）。

unittest实例

下面我们通过一些实例来更好地认识一下unittest。

们先来准备一些待测方法：

mathfunc.py

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

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

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

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

简单示例

接下来我们为这些方法写一个测试:

test_mathfunc.py

```
# -*- coding: utf-8 -*-

import unittest
from mathfunc import *


class TestMathFunc(unittest.TestCase):
    """Test mathfuc.py"""

    def test_add(self):
        """Test method add(a, b)"""
        self.assertEqual(3, add(1, 2))
        self.assertNotEqual(3, add(2, 2))

    def test_minus(self):
        """Test method minus(a, b)"""
        self.assertEqual(1, minus(3, 2))

    def test_multi(self):
        """Test method multi(a, b)"""
        self.assertEqual(6, multi(2, 3))

    def test_divide(self):
        """Test method divide(a, b)"""
        self.assertEqual(2, divide(6, 3))
        self.assertEqual(2.5, divide(5, 2))

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

执行结果:

```
.F..
======================================================================
FAIL: test_divide (__main__.TestMathFunc)
Test method divide(a, b)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:/py/test_mathfunc.py", line 26, in test_divide
    self.assertEqual(2.5, divide(5, 2))
AssertionError: 2.5 != 2

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=1)
```

能够看到一共运行了4个测试，失败了1个，并且给出了失败原因，`2.5 != 2` 也就是说我们的divide方法是有问题的。

这就是一个简单的测试，有几点需要说明的：

1. 在第一行给出了每一个用例执行的结果的标识，成功是 `.`，失败是 `F`，出错是 `E`，跳过是 `S`。从上面也可以看出，测试的执行跟方法的顺序没有关系，test_divide写在了第4个，但是却是第2个执行的。
2. 每个测试方法均以 `test` 开头，否则是不被unittest识别的。
3. 在unittest.main()中加 `verbosity` 参数可以控制输出的错误报告的详细程度，默认是 `1`，如果设为 `0`，则不输出每一用例的执行结果，即没有上面的结果中的第1行；如果设为 `2`，则输出详细的执行结果，如下：

```
test_add (__main__.TestMathFunc)
Test method add(a, b) ... ok
test_divide (__main__.TestMathFunc)
Test method divide(a, b) ... FAIL
test_minus (__main__.TestMathFunc)
Test method minus(a, b) ... ok
test_multi (__main__.TestMathFunc)
Test method multi(a, b) ... ok

======================================================================
FAIL: test_divide (__main__.TestMathFunc)
Test method divide(a, b)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:/py/test_mathfunc.py", line 26, in test_divide
    self.assertEqual(2.5, divide(5, 2))
AssertionError: 2.5 != 2

----------------------------------------------------------------------
Ran 4 tests in 0.002s

FAILED (failures=1)
```

可以看到，每一个用例的详细执行情况以及用例名，用例描述均被输出了出来（在测试方法下加代码示例中的""" "Doc String"""，在用例执行时，会将该字符串作为此用例的描述，**加合适的注释能够使输出的测试报告更加便于阅读**）

组织TestSuite

上面的代码示例了如何编写一个简单的测试，但有两个问题，我们怎么控制用例执行的顺序呢？（这里的示例中的几个测试方法并没有一定关系，但之后你写的用例可能会有先后关系，需要先执行方法A，再执行方法B），我们就要用到TestSuite了。我们**添加到TestSuite中的case是会按照添加的顺序执行的**。

问题二是我们现在只有一个测试文件，我们直接执行该文件即可，但如果有多个测试文件，怎么进行组织，总不能一个个文件执行吧，答案也在TestSuite中。

下面来个例子：

在文件夹中我们再新建一个文件，**test_suite.py**：

```
# -*- coding: utf-8 -*-

import unittest
from test_mathfunc import TestMathFunc

if __name__ == '__main__':
    suite = unittest.TestSuite()

    tests = [TestMathFunc("test_add"), TestMathFunc("test_minus"), TestMathFunc("test
    suite.addTests(tests)

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

执行结果：

```
test_add (test_mathfunc.TestMathFunc)
Test method add(a, b) ... ok
test_minus (test_mathfunc.TestMathFunc)
Test method minus(a, b) ... ok
```

```
test_divide (test_mathfunc.TestMathFunc)
Test method divide(a, b) ... FAIL

======================================================================
FAIL: test_divide (test_mathfunc.TestMathFunc)
Test method divide(a, b)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\py\test_mathfunc.py", line 26, in test_divide
    self.assertEqual(2.5, divide(5, 2))
AssertionError: 2.5 != 2

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```


「以看到，执行情况跟我们预料的一样：执行了三个case，并且顺序是按照我们添加进suite的顺序执行的。

:面用了TestSuite的 `addTests()` 方法，并直接传入了TestCase列表，我们还可以：

```
# 直接用addTest方法添加单个TestCase
suite.addTest(TestMathFunc("test_multi"))

# 用addTests + TestLoader
# loadTestsFromName()，传入'模块名.TestCase名'
suite.addTests(unittest.TestLoader().loadTestsFromName('test_mathfunc.TestMathFunc'))
suite.addTests(unittest.TestLoader().loadTestsFromNames(['test_mathfunc.TestMathFunc

# loadTestsFromTestCase()，传入TestCase
suite.addTests(unittest.TestLoader().loadTestsFromTestCase(TestMathFunc))
```

注意，用TestLoader的方法是无法对case进行排序的，同时，suite中也可以套suite。

将结果输出到文件中

用例组织好了，但结果只能输出到控制台，这样没有办法查看之前的执行记录，我们想将结果输出到文件。很简单，看示例：

修改**test_suite.py**：

```
# -*- coding: utf-8 -*-

import unittest
from test_mathfunc import TestMathFunc

if __name__ == '__main__':
    suite = unittest.TestSuite()
    suite.addTests(unittest.TestLoader().loadTestsFromTestCase(TestMathFunc))

    with open('UnittestTextReport.txt', 'a') as f:
        runner = unittest.TextTestRunner(stream=f, verbosity=2)
        runner.run(suite)
```

执行此文件，可以看到，在同目录下生成了**UnittestTextReport.txt**，所有的执行报告均输出到了此文件中，这下我们便有了txt格式的测试报告了。

test fixture之setUp() tearDown()

上面整个测试基本跑了下来，但可能会遇到点特殊的情况：如果我的测试需要在每次执行之前准备环境，或者在每次执行完之后需要进行一些清理怎么办？比如执行前需要连接数据库，执行完成之后需要还原数据、断开连接。总不能每个测试方法中都添加准备环境、清理环境的代码吧。

这就要涉及到我们之前说过的test fixture了，修改**test_mathfunc.py**：

```
# -*- coding: utf-8 -*-

import unittest
from mathfunc import *


class TestMathFunc(unittest.TestCase):
    """Test mathfuc.py"""

    def setUp(self):
```

```
        print "do something before test.Prepare environment."

    def tearDown(self):
        print "do something after test.Clean up."

    def test_add(self):
        """Test method add(a, b)"""
        print "add"
        self.assertEqual(3, add(1, 2))
        self.assertNotEqual(3, add(2, 2))

    def test_minus(self):
        """Test method minus(a, b)"""
        print "minus"
        self.assertEqual(1, minus(3, 2))

    def test_multi(self):
        """Test method multi(a, b)"""
        print "multi"
        self.assertEqual(6, multi(2, 3))

    def test_divide(self):
        """Test method divide(a, b)"""
        print "divide"
        self.assertEqual(2, divide(6, 3))
        self.assertEqual(2.5, divide(5, 2))
```

我们添加了 `setUp()` 和 `tearDown()` 两个方法（其实是重写了TestCase的这两个方法），这两个方法在每个测试方法执行前以及执行后执行一次，setUp用来为测试准备环境，tearDown用来清理环境，已备之后的测试。

我们再执行一次：

```
test_add (test_mathfunc.TestMathFunc)
Test method add(a, b) ... ok
test_divide (test_mathfunc.TestMathFunc)
Test method divide(a, b) ... FAIL
test_minus (test_mathfunc.TestMathFunc)
Test method minus(a, b) ... ok
test_multi (test_mathfunc.TestMathFunc)
Test method multi(a, b) ... ok

======================================================================
FAIL: test_divide (test_mathfunc.TestMathFunc)
Test method divide(a, b)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\py\test_mathfunc.py", line 36, in test_divide
    self.assertEqual(2.5, divide(5, 2))
AssertionError: 2.5 != 2

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=1)
do something before test.Prepare environment.
add
do something after test.Clean up.
do something before test.Prepare environment.
divide
do something after test.Clean up.
do something before test.Prepare environment.
minus
do something after test.Clean up.
do something before test.Prepare environment.
multi
do something after test.Clean up.
```

可以看到setUp和tearDown在每次执行case前后都执行了一次。

如果想要在所有case执行之前准备一次环境，并在所有case执行结束之后再清理环境，我们可以用 `setUpClass()` 与 `tearDownClass()`：

```
...

class TestMathFunc(unittest.TestCase):
    """Test mathfuc.py"""
```

```
5
6       @classmethod
7       def setUpClass(cls):
8           print "This setUpClass() method only called once."
9
10      @classmethod
11      def tearDownClass(cls):
12          print "This tearDownClass() method only called once too."
13
14  ...
```

执行结果如下：

```
1  ...
2  This setUpClass() method only called once.
3  do something before test.Prepare environment.
4  add
5  do something after test.Clean up.
6  ...
7  do something before test.Prepare environment.
8  multi
9  do something after test.Clean up.
10 This tearDownClass() method only called once too.
```

可以看到setUpClass以及tearDownClass均只执行了一次。

跳过某个case

如果我们临时想要跳过某个case不执行怎么办？unittest也提供了几种方法：

1. skip装饰器

```
1  ...
2
3  class TestMathFunc(unittest.TestCase):
4      """Test mathfuc.py"""
5
6      ...
7
8      @unittest.skip("I don't want to run this case.")
9      def test_divide(self):
10         """Test method divide(a, b)"""
11         print "divide"
12         self.assertEqual(2, divide(6, 3))
13         self.assertEqual(2.5, divide(5, 2))
```

执行：

```
1  ...
2  test_add (test_mathfunc.TestMathFunc)
3  Test method add(a, b) ... ok
4  test_divide (test_mathfunc.TestMathFunc)
5  Test method divide(a, b) ... skipped "I don't want to run this case."
6  test_minus (test_mathfunc.TestMathFunc)
7  Test method minus(a, b) ... ok
8  test_multi (test_mathfunc.TestMathFunc)
9  Test method multi(a, b) ... ok
10
11 ----------------------------------------------------------------------
12 Ran 4 tests in 0.000s
13
14 OK (skipped=1)
```

可以看到总的test数量还是4个，但divide()方法被skip了。

skip装饰器一共有三个 `unittest.skip(reason)`、`unittest.skipIf(condition, reason)`、`unittest.skipUnless(condition, reason)`，skip无条件跳过，skipIf当condition为True时跳过，skipUnless当condition为False时跳过。

1. TestCase.skipTest()方法

```
1  ...
2
3  class TestMathFunc(unittest.TestCase):
```

```
    """Test mathfuc.py"""

    ...

    def test_divide(self):
        """Test method divide(a, b)"""
        self.skipTest('Do not run this.')
        print "divide"
        self.assertEqual(2, divide(6, 3))
        self.assertEqual(2.5, divide(5, 2))
```

输出：

```
...
test_add (test_mathfunc.TestMathFunc)
Test method add(a, b) ... ok
test_divide (test_mathfunc.TestMathFunc)
Test method divide(a, b) ... skipped 'Do not run this.'
test_minus (test_mathfunc.TestMathFunc)
Test method minus(a, b) ... ok
test_multi (test_mathfunc.TestMathFunc)
Test method multi(a, b) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK (skipped=1)
```

效果跟上面的装饰器一样，跳过了divide方法。

进阶——用HTMLTestRunner输出漂亮的HTML报告

我们能够输出txt格式的文本执行报告了，但是文本报告太过简陋，是不是想要更加高大上的HTML报告？但unittest自己可没有带HTML报告，我们只能求助于外部的库了。

HTMLTestRunner是一个第三方的unittest HTML报告库，首先我们下载**HTMLTestRunner.py**，并放到当前目录下，或者你的'C:\Python27\Lib'下，就可以导入运行了。

下载地址：

> 官方原版：http://tungwaiyip.info/software/HTMLTestRunner.html
> 灰蓝修改版：HTMLTestRunner.py(已调整格式，中文显示)

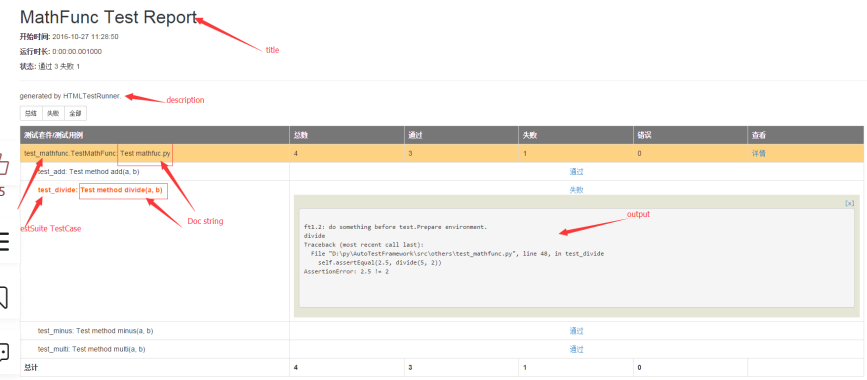
修改我们的 `test_suite.py`：

```
# -*- coding: utf-8 -*-

import unittest
from test_mathfunc import TestMathFunc
from HTMLTestRunner import HTMLTestRunner

if __name__ == '__main__':
    suite = unittest.TestSuite()
    suite.addTests(unittest.TestLoader().loadTestsFromTestCase(TestMathFunc))

    with open('HTMLReport.html', 'w') as f:
        runner = HTMLTestRunner(stream=f,
                                title='MathFunc Test Report',
                                description='generated by HTMLTestRunner.',
                                verbosity=2
                                )
        runner.run(suite)
```

这样，在执行时，在控制台我们能够看到执行情况，如下：

```
ok test_add (test_mathfunc.TestMathFunc)
F  test_divide (test_mathfunc.TestMathFunc)
ok test_minus (test_mathfunc.TestMathFunc)
ok test_multi (test_mathfunc.TestMathFunc)

Time Elapsed: 0:00:00.001000
```

并且输出了HTML测试报告，`HTMLReport.html`，如图：



下漂亮的HTML报告也有了。其实你能发现，HTMLTestRunner的执行方法跟TextTestRunner很相似，你可以我上面的示例对比一下，就是把类图中的runner换成了HTMLTestRunner，并将TestResult用HTML的形式展现出来，如果你研究够深，可以写自己的runner，生成更复杂更漂亮的报告。

总结一下：

1. unittest是Python自带的单元测试框架，我们可以用其来作为我们自动化测试框架的用例组织执行框架。
2. unittest的流程：写好TestCase，然后由TestLoader加载TestCase到TestSuite，然后由TextTestRunner来运行TestSuite，运行的结果保存在TextTestResult中，我们通过命令行或者unittest.main()执行时，main会调用TextTestRunner中的run来执行，或者我们可以直接通过TextTestRunner来执行用例。
3. 一个class继承unittest.TestCase即是一个TestCase，其中以 `test` 开头的方法在load时被加载为一个真正的TestCase。
4. verbosity参数可以控制执行结果的输出， `0` 是简单报告、 `1` 是一般报告、 `2` 是详细报告。
5. 可以通过addTest和addTests向suite中添加case或suite，可以用TestLoader的loadTestsFrom__()方法。
6. 用 `setUp()` 、 `tearDown()` 、 `setUpClass()` 以及 `tearDownClass()` 可以在用例执行前布置环境，以及在用例执行后清理环境
7. 我们可以通过skip，skipIf，skipUnless装饰器跳过某个case，或者用TestCase.skipTest方法。
8. 参数中加stream，可以将报告输出到文件：可以用TextTestRunner输出txt报告，以及可以用HTMLTestRunner输出html报告。

我们这里没有讨论命令行的使用以及模块级别的fixture，感兴趣的同学可以自行搜索资料学习。