

软件测试教程 自动化测试之unittest框架

本节课我们来学习单元测试框架unittest，这里的单元测试指的是对最小的软件设计单元（模块）进行验证，在UI自动化测试里面，我们的单元测试主要针对UI界面的功能进行自动化测试。所以这里大家要注意不要和Java的JUnit单元测试框架搞混；

本课程将讲解unittest框架以下几个方面：

unittest框架解析

批量执行脚本

unittest断言

HTML报告生成

异常捕捉与错误截图

数据驱动

unittest框架解析

unittest 是python 的单元测试框架，它主要有以下作用：

提供用例组织与执行：当你的测试用例只有几条时，可以不必考虑用例的组织，但是，当测试用例达到成百上千条时，大量的测试用例堆砌在一起，就产生了扩展性与维护性等问题，此时需要考虑用例的规范与组织问题了。单元测试框架就是来解决这个问题的。

提供丰富的比较方法：在用例执行完之后都需要将实际结果与预期结果进行比较（断言），从而断定用例是否可以顺利通过。单元测试一般会提供丰富的断言方法。例如，判断相等/不相等、包含/不包含、True/False等断言方法。

提供丰富的日志：当测试用例执行失败时能抛出清晰的失败原因，当所有用例执行完成后能提供丰富的执行结果。例如，总的执行时间、失败用例数，成功用例数等。

unittest里面有四个很重要的概念，test fixture, test case, test suite, test runner。

- Test Fixture

对一个测试用例环境的搭建和销毁，就是一个fixture，通过覆盖setUp()和tearDown()方法来实现。

setUp()方法可以进行测试环境的搭建，比如获取待测试浏览器的驱动，或者如果测试中需要访问数据库，那么可以在setUp()中通过建立数据库连接来进行初始化。

tearDown()方法进行环境的销毁，可以进行关闭浏览器，关闭数据库连接，清除数据库中产生的数据等操作；

- Test Case

一个TestCase的实例就是一个测试用例。测试用例就是一个完整的测试流程，包括测试前准备环境的搭建（setUp）、实现测试过程的代码，以及测试后环境的还原（tearDown）。单元测试（unit test）的本质就在这里，一个测试用例就是一个完整的测试单元，可以对某一个功能进行验证。

- Test Suite

一个功能的验证往往需要多个测试用例，可以把多个测试用例集合在一起执行，这个套件TestSuite的概念。Test Suit用来将多个测试用例组装在一起；

- Test Runner

测试的执行也是非常重要的一个概念，在unittest框架中，通过TextTestRunner类提供的run()方法来执行test suite/test case。

下面为一个使用了unittest框架的脚本：

```
from selenium import webdriver
import unittest
import time
import os
from selenium.common.exceptions import NoAlertPresentException
from selenium.common.exceptions import NoSuchElementException

class Baidu1(unittest.TestCase):
    def setUp(self):
        print("-----setUp-----")
        self.driver = webdriver.Chrome()
        self.url = "https://www.baidu.com/"
        self.driver.maximize_window()
        time.sleep(3)

    def tearDown(self):
        print("-----tearDown-----")
        self.driver.quit()

    def test_hao(self):
        driver = self.driver
        url = self.url
        driver.get(url)
        driver.find_element_by_link_text("hao123").click()
        time.sleep(6)

    def test_hbaidu(self):
        driver = self.driver
        url = self.url
        driver.get(url)
        driver.find_element_by_id("kw").send_keys("突如其来的假期")
        driver.find_element_by_id("su").submit()
        time.sleep(5)
        print(driver.title)
        # self.assertEqual(driver.title, "百度一下_百度搜索", msg="不相等")
        # self.assertTrue("beautiful"=="beauty", msg="Not Equal!")
        time.sleep(6)

    def saveScreenAsPhoto(self, driver, file_name):
        if not os.path.exists("./image"):
            os.makedirs("./image")
        now = time.strftime("%Y%m%d-%H%M%S", time.localtime(time.time()))
        driver.get_screenshot_as_file("./image/" + now + "-" + file_name)
        time.sleep(3)

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

这个脚本中的类 Baidu1 继承了unittest.TestCase类，所以它使用了unittest框架来组织测试用例 (TestCase)。

setUp() 和 tearDown() 是unittest框架中的测试固件

以test_开头命名的方法，是测试方法，在运行整个类的时候会默认执行。

unittest提供了全局的main()方法，使用它可以方便地将一个单元测试模块变成可以直接运行的测试脚本。main()方法搜索所有包含在该模块中以“test”命名的测试方法，并自动执行他们。

批量执行脚本

构建测试套件

当我们增加了被测试功能和相应的测试用例之后，我们就需要把多个测试用例组织在一起执行，那 unittest框架是如何扩展和组织新增的测试用例的呢？它使用的就是上文中提到的测试套件Test Suite

假设我们已经编写了testbaidu1.py, testbaidu2.py两个文件，那么我们怎么同时执行这两个文件呢？

testbaidu1.py

```
from selenium import webdriver
import unittest
import time
import os
from selenium.common.exceptions import NoAlertPresentException
from selenium.common.exceptions import NoSuchElementException

class Baidu1(unittest.TestCase):

    def setup(self):
        self.driver = webdriver.Chrome()
        self.url = "https://www.baidu.com/"
        self.driver.maximize_window()
        time.sleep(3)

    def tearDown(self):
        self.driver.quit()

    def test_hao(self):
        driver = self.driver
        url = self.url
        driver.get(url)
        driver.find_element_by_link_text("hao123").click()
        time.sleep(6)

    def test_hbaidu(self):
        driver = self.driver
        url = self.url
        driver.get(url)
        driver.find_element_by_id("kw").send_keys("突如其来的假期")
        driver.find_element_by_id("su").submit()
        time.sleep(5)
        print(driver.title)
        time.sleep(6)
```



```
def saveScreenAsPhoto(self, driver, file_name):
    if not os.path.exists("./image"):
        os.makedirs("./image")
    now = time.strftime("%Y%m%d-%H%M%S", time.localtime(time.time()))
    driver.get_screenshot_as_file("./image/" + now + "-" + file_name)
    time.sleep(3)

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

testbaidu2.py

```
# -*- coding: utf-8 -*-
from selenium import webdriver
import unittest
import time
from selenium.common.exceptions import NoAlertPresentException
from selenium.common.exceptions import NoSuchElementException

class Baidu2 (unittest.TestCase) :
    def setUp(self):
        self.driver = webdriver.Chrome()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.baidu.com/"
        self.driver.maximize_window()
        self.verificationErrors=[]
        self.accept_next_alert = True
    def tearDown(self):
        self.driver.quit()
        self.assertEqual([], self.verificationErrors)

    def test_hao(self):
        driver = self.driver
        driver.get(self.base_url)
        driver.find_element_by_link_text("新闻").click()
        time.sleep(6)
        self.assertTrue("123" == "1234", msg="not true")
        time.sleep(3)

    def test_baidusearch(self):
        driver = self.driver
        driver.get(self.base_url)
        driver.find_element_by_id("kw").clear()
        driver.find_element_by_id("kw").send_keys(u"庆余年")
        driver.find_element_by_id("su").click()
        time.sleep(6)

    def is_element_present(self, how, what):
        try:
            self.driver.find_element(by=how, value=what)
        except NoSuchElementException as e:
            return False
        return True

    def is_alert_present(self):
        try:
            self.driver.switch_to.alert
```



```

except NoAlertPresentException as e:
    return False
return True

def close_alert_and_get_its_text(self):
    try:
        alert = self.driver.switch_to.alert
        alert_text = alert.text
        if self.accept_next_alert:
            alert.accept()
        else:
            alert.dismiss()
        return alert_text
    finally: self.accept_next_alert = True
if __name__ == "__main__":
    unittest.main(verbosity=2)

```

addTest()

TestSuite类的addTest()方法可以把不同的测试类中的测试方法组装到测试套件中，但是addTest()一次只能把一个类里面的一个测试方法组装到测试套件中。方式如下：

将testbaidu1.py、testbaidu2.py中的测试方法放到一个测试套件中，在testsuite.py中实现。

testsuite.py

```

import unittest
from src0716 import testbaidu1
from src0716 import testbaidu2

def createsuite():
    #addTest
    suite = unittest.TestSuite()
    suite.addTest(testbaidu1.Baidu1("test_hao"))
    suite.addTest(testbaidu1.Baidu1("test_hbaidu"))
    suite.addTest(testbaidu2.Baidu2("test_hao"))
    suite.addTest(testbaidu2.Baidu2("test_baidusearch"))
    return suite

if __name__=="__main__":
    suite = createsuite()
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)

```

但是上述做法有两个不方便的地方，阻碍脚本的快速执行，必须每次修改testsuite.py：

- 1) 需要导入所有的相关的py文件，比如 import testbaidu1，每新增一个脚本就需要导入一个
- 2) addTest一次只能增加一个测试方法，如果一个py文件中有10个测试方法，如果都要组装到测试套件中，就需要增加10次

makeSuite () 和TestLoader() 的应用

在unittest框架中提供了makeSuite() 的方法，makeSuite可以实现把测试用例类内所有的测试case组成的测试套件TestSuite，unittest 调用makeSuite的时候，只需要把测试类名称传入即可。

TestLoader 用于创建类和模块的测试套件，一般的情况下，使
TestLoader().loadTestsFromTestCase(TestCase) 来加载测试类。

runall.py

```
# -*- coding: utf-8 -*-
import unittest, csv
import os, sys
import time
import testbaidu1
import testbaidu2

#手工添加案例到套件,
def createsuite():

    suite = unittest.TestSuite()
    #将测试用例加入到测试容器（套件）中
    suite.addTest(unittest.makeSuite(testbaidu1.Baidu1))
    suite.addTest(unittest.makeSuite(testbaidu2.Baidu2))
    return suite
    '''

    suite1 = unittest.TestLoader().loadTestsFromTestCase(testbaidu1.Baidu1)
    suite2 = unittest.TestLoader().loadTestsFromTestCase(testbaidu2.Baidu2)
    suite = unittest.TestSuite([suite1, suite2])
    return suite
    '''

if __name__=="__main__":
    suite=createsuite()
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)
```

经过makeSuite () 和TestLoader () 的引入，我们不用一个py文件测试类，只需要导入一次即可。

那么能不能测试类也不用每次添加指定呢？

discover () 的应用

discover 是通过递归的方式到其子目录中从指定的目录开始，找到所有测试模块并返回一个包含它们对象的TestSuite，然后进行加载与模式匹配唯一的测试文件，discover 参数分别为
discover(dir, pattern, top_level_dir=None)

runall.py--注意路径

```
# -*- coding: utf-8 -*-
import unittest, csv
import os, sys
import time

#手工添加案例到套件,
def createsuite():

    discover=unittest.defaultTestLoader.discover('../test',pattern='test*.py',top_level_dir=None)
    print discover
    return discover

if __name__=="__main__":
```



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

用例的执行顺序

unittest 框架默认加载测试用例的顺序是根据ASCII 码的顺序，数字与字母的顺序为： 0~9,A~Z,a~z 。

所以， TestAdd 类会优先于TestBdd 类被发现， test_aaa() 方法会优先于test_ccc() 被执行。对于测试目录与测试文件来说， unittest 框架同样是按照这个规则来加载测试用例。

addTest () 方法按照增加顺序来执行。

忽略用例执行

```
@unittest.skip(u'The function was canceled, neglects to perform thecase')
```

```
# -*- coding: utf-8 -*-
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
from selenium.common.exceptions import NoAlertPresentException
import unittest, time, re

class Baidu1(unittest.TestCase):
    #test fixture, 初始化环境
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.baidu.com/"
        selfverificationErrors = []
        self.accept_next_alert = True

    @unittest.skip("skipping")
    def test_baidusearch(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_id("kw").click()
        driver.find_element_by_id("kw").clear()
        driver.find_element_by_id("kw").send_keys(u"测试")
        driver.find_element_by_id("su").click()
        driver.find_element_by_id("su").click()

    def test_hao(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_link_text("hao123").click()
        self.assertEqual(u"hao123_上网从这里开始", driver.title)

    #判断element是否存在, 可删除
    def is_element_present(self, how, what):
        try: self.driver.find_element(by=how, value=what)
        except NoSuchElementException as e: return False
```

```

        return True
#判断alert是否存在, 可删除
def is_alert_present(self):
    try: self.driver.switch_to_alert()
    except NoAlertPresentException as e: return False
    return True
#关闭alert, 可删除
def close_alert_and_get_its_text(self):
    try:
        alert = self.driver.switch_to_alert()
        alert_text = alert.text
        if self.accept_next_alert:
            alert.accept()
        else:
            alert.dismiss()
        return alert_text
    finally: self.accept_next_alert = True
#test fixture, 清除环境
def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

if __name__ == "__main__":
#执行用例
    unittest.main()

```

unittest断言

自动化的测试中，对于每个单独的case来说，一个case的执行结果中，必然会有期望结果与实际结果，来判断该case是通过还是失败，在unittest的库中提供了大量的实用方法来检查预期值与实际值，来验证case的结果，一般来说，检查条件大体分为等价性，逻辑比较以及其他，如果给定的断言通过，测试会继续执行到下一行的代码，如果断言失败，对应的case测试会立即停止或者生成错误信息(一般打印错误信息即可)，但是不要影响其他的case执行。

unittest 的单元测试库提供了标准的xUnit 断言方法。下面是一些常用的断言

序号	断言方法	断言描述
1	assertEqual(arg1, arg2, msg=None)	验证arg1==arg2, 不等则fail
2	assertNotEqual(arg1, arg2, msg=None)	验证arg1 != arg2, 相等则fail
3	assertTrue(expr, msg=None)	验证expr是true, 如果为false, 则fail
4	assertFalse(expr, msg=None)	验证expr是false, 如果为true, 则fail
5	assertIs(arg1, arg2, msg=None)	验证arg1、arg2是同一个对象，不是则fail
6	assertIsNot(arg1, arg2, msg=None)	验证arg1、arg2不是同一个对象，是则fail
7	assertIsNone(expr, msg=None)	验证expr是None, 不是则fail
8	assertIsNotNone(expr, msg=None)	验证expr不是None, 是则fail
9	assertIn(arg1, arg2, msg=None)	验证arg1是arg2的子串, 不是则fail

序号	断言方法	断言描述
11	assertIsInstance(obj, cls, msg=None)	验证obj是cls的实例，不是则fail
12	assertNotIsInstance(obj, cls, msg=None)	验证obj不是cls的实例，是则fail

举例：

```
self.assertEqual("admin", driver.find_element_by_link_text("admin").text)
```

HTML报告生成

脚本执行完毕之后，还需要看到HTML报告，下面我们就通过HTMLTestRunner.py 来生成测试报告。HTMLTestRunner支持python2.7。python3可以参见<http://blog.51cto.com/hzqldjb/1590802>来进行修改。

HTMLTestRunner.py 文件，下载地址：<http://tungwaiyip.info/software/HTMLTestRunner.html>

下载后将其放在testcase目录中去或者放入...\\Python27\\Lib 目录下（windows）。

修改runall.py

```
# -*- coding: utf-8 -*-
import unittest, csv
import os,sys
import time
import HTMLTestRunner

#手工添加案例到套件,
def createsuite():

discover=unittest.defaultTestLoader.discover('../test',pattern='test*.py',top_level_dir=None)
    print discover
    return discover

if __name__=="__main__":
    curpath=sys.path[0]
    #取当前时间
    now=time.strftime("%Y-%m-%d-%H %M %S",time.localtime(time.time()))

    if not os.path.exists(curpath+'/resultreport'):
        os.makedirs(curpath+'/resultreport')

    filename=curpath+'/resultreport/'+now+'resultreport.html'
    with open(filename,'wb') as fp:
        #出html报告
        runner=HTMLTestRunner.HTMLTestRunner(stream=fp,title=u'测试报告',description=u'用例执行情况',verbosity=2)
        suite=createsuite()
        runner.run(suite)
```

异常捕捉与错误截图

用例不可能每一次运行都成功，肯定运行时候有不成功的时候。如果可以捕捉到错误，并且把错误截图保存，这将是一个非常棒的功能，也会给我们错误定位带来方便。

例如编写一个函数，关键语句为driver.get_screenshot_as_file：

```
def savescreenshot(self, driver, file_name):
    if not os.path.exists('./image'):
        os.makedirs('./image')
    now=time.strftime("%Y%m%d-%H%M%S",time.localtime(time.time()))
    #截图保存
    driver.get_screenshot_as_file('./image/'+now+'-'+file_name)
    time.sleep(1)
```

一个引用的例子：

testscreenshot.py

```
# -*- coding: utf-8 -*-
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
from selenium.common.exceptions import NoAlertPresentException
import unittest, time, re
import os

class Baidu1(unittest.TestCase):
#test fixture, 初始化环境
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.baidu.com/"
        selfverificationErrors = []
        self.accept_next_alert = True

    #测试用例，必须以test开头

    def test_hao(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_link_text("hao123").click()
        time.sleep(2)
        try:
            self.assertEqual(u'hao_上网从这里开始', driver.title)
        except:
            self.savescreenshot(driver, 'hao.png')

    #判断element是否存在，可删除
    def is_element_present(self, how, what):
        try: self.driver.find_element(by=how, value=what)
        except NoSuchElementException as e: return False
        return True

    #判断alert是否存在，可删除
```



```

def is_alert_present(self):
    try: self.driver.switch_to_alert()
    except NoAlertPresentException as e: return False
    return True
#关闭alert, 可删除
def close_alert_and_get_its_text(self):
    try:
        alert = self.driver.switch_to_alert()
        alert_text = alert.text
        if self.accept_next_alert:
            alert.accept()
        else:
            alert.dismiss()
        return alert_text
    finally: self.accept_next_alert = True
#test fixture, 清除环境
def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

def savescreenshot(self,driver,file_name):
    if not os.path.exists('./image'):
        os.makedirs('./image')
    now=time.strftime("%Y%m%d-%H%M%S",time.localtime(time.time()))
    #截图保存
    driver.get_screenshot_as_file('./image/' + now + '-' + file_name)
    time.sleep(1)

if __name__ == "__main__":
    #执行用例
    unittest.main()
    ...

```

可以增加**verbosity**参数，例如**unittest.main(verbosity=2)**
在主函数中，直接调用**main()**，在**main**中加入**verbosity=2**，这样测试的结果就会显示的更加详细。
这里的**verbosity** 是一个选项，表示测试结果的信息复杂度，有三个值：
0 (静默模式)：你只能获得总的测试用例数和总的结果比如总共100个失败, 20 成功80
1 (默认模式)：非常类似静默模式只是在每个成功的用例前面有个“.” 每个失败的用例前面有个“F”
2 (详细模式)：测试结果会显示每个测试用例的所有相关的信息
 ...

数据驱动

之前我们的case都是数据和代码在一起编写。考虑如下场景：

需要多次执行一个案例，比如baidu搜索，分别输入中文、英文、数字等进行搜索，这时候需要编写3个案例吗？有没有版本一次运行？

python 的**unittest** 没有自带数据驱动功能。所以如果使用**unittest**，同时又想使用数据驱动，那么就可以使用**DDT**来完成。

ddt的安装： <http://ddt.readthedocs.io/en/latest/> <https://github.com/txels/ddt>

```

pip install ddt
python setup.py install

```

ddt使用方法:

参考文档: <http://ddt.readthedocs.io/en/latest/>

dd.ddt:

装饰类, 也就是继承自TestCase的类。

ddt.data:

装饰测试方法。参数是一系列的值。

ddt.file_data:

装饰测试方法。参数是文件名。文件可以是json 或者 yaml类型。

注意, 如果文件以".yml"或者".yaml"结尾, ddt会作为yaml类型处理, 其他所有文件都会作为json文件处理。

如果文件中是列表, 每个列表的值会作为测试用例参数, 同时作为测试用例方法名后缀显示。

如果文件中是字典, 字典的key会作为测试用例方法的后缀显示, 字典的值会作为测试用例参数。

ddt.unpack:

传递的是复杂的数据结构时使用。比如使用元组或者列表, 添加unpack之后, ddt会自动把元组或者列表对应到多个参数上。字典也可以这样处理。

下面看一个样例:

test_data_list.json:

```
[  
    "Hello",  
    "Goodbye"  
]
```

Testddt.py: 目录下建data文件夹, 并建一个test_baidu_data.csv文件 (用execl (最好用TXT文件) 另存为csv)

```
#-*- coding: utf-8 -*-  
from selenium import webdriver  
from selenium.webdriver.common.by import By  
from selenium.webdriver.common.keys import Keys  
from selenium.webdriver.support.ui import Select  
from selenium.common.exceptions import NoSuchElementException  
from selenium.common.exceptions import NoAlertPresentException  
import unittest, time, re  
import os,sys, csv  
from ddt import ddt, data, unpack ,file_data  
  
def getcsv(file_name):  
    rows=[]  
    path=sys.path[0].replace('\\test','')  
    print path  
  
    with open(path+'/data/'+file_name,'rb') as f:  
        readers=csv.reader(f,delimiter=',', quotechar='|')  
        next(readers,None)  
        for row in readers:  
            temprows=[]
```

```
        for i in row:
            temprows.append(i.decode('gbk'))
        rows.append(temprows)
        return rows

#引入ddt
@ddt
class Testddt(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.baidu.com"
        selfverificationErrors = []
        self.accept_next_alert = True

    #测试用例，必须以test开头
    #增加ddt数据
    #@data('selenium',u'测试中文','9999999999')
    #@data(2,3,4)
    #单变更时不使用unpack
    #@data([3, 2], [4, 3], [5, 3])
    @data(*getCsv('test_baidu_data.csv'))
    #使用file_data需要在cmd窗口下运行，否则找不到文件
    #@file_data('test_data_list.json')
    @unpack

    def test_hao(self,value,expected_value):
        #def test_hao(self,value):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_id("kw").clear()
        driver.find_element_by_id("kw").send_keys(value)
        driver.find_element_by_id("su").click()
        time.sleep(2)
        self.assertEqual(expected_value, driver.title)
        print expected_value
        print driver.title
    #判断element是否存在，可删除
    def is_element_present(self, how, what):
        try: self.driver.find_element(by=how, value=what)
        except NoSuchElementException as e: return False
        return True
    #判断alert是否存在，可删除
    def is_alert_present(self):
        try: self.driver.switch_to_alert()
        except NoAlertPresentException as e: return False
        return True
    #关闭alert，可删除
    def close_alert_and_get_its_text(self):
        try:
            alert = self.driver.switch_to_alert()
            alert_text = alert.text
            if self.accept_next_alert:
                alert.accept()
            else:
                alert.dismiss()
            return alert_text
        finally: self.accept_next_alert = True
    #test fixture, 清除环境
```

```
def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

def savescreenshot(self,driver,file_name):
    if not os.path.exists('./image'):
        os.makedirs('./image')
    now=time.strftime("%Y%m%d-%H%M%S",time.localtime(time.time()))
    #截图保存
    driver.get_screenshot_as_file('./image/'+now+'-'+file_name)
    time.sleep(1)

if __name__ == "__main__":
#执行用例
    unittest.main()

'''
```

可以增加**verbosity**参数，例如**unittest.main(verbosity=2)**
在主函数中，直接调用**main()**，在**main**中加入**verbosity=2**，这样测试的结果就会显示的更加详细。
这里的**verbosity** 是一个选项，表示测试结果的信息复杂度，有三个值：
0 (静默模式)：你只能获得总的测试用例数和总的结果比如总共100个失败,20 成功80
1 (默认模式)：非常类似静默模式只是在每个成功的用例前面有个“.”，每个失败的用例前面有个“F”
2 (详细模式)：测试结果会显示每个测试用例的所有相关的信息
'''