Carib.J.Sci.Tech

Demonstration of Unit Testing Python Programs

Authors & Affiliation:

S N V Satyanarayana P,
Assistant Professor, Swarnandhra Institute of Engineering & Technology, Seetharampuram, Narsapur, W. G. Dt., Andhra Pradesh, India.

Corresponding Author:

S N V Satyanarayana P

ISSN 0799-3757

http://caribjscitech.com/

Abstract:

Software Testing plays the most prominent role in ensuring the quality of software. Unit testing tests an individual program with the intent of finding errors in it before it is integrated with the other programs of the software. Like JUnit for Java, most of the popular programming languages have been implemented such unit testing frameworks for testing individual programs before integration. In this paper, I presented the details of unit testing framework (doctest, unittest) supported by Python and process of unit testing python programs. I also showcased writing test cases and executing them with illustrations and results.

Keywords: Software Testing, Python, Unit Testing, doctest, unittest, Test case.

1. Introduction:

This paper mainly focuses on demonstrating the practical usage of unit test framework in Python. Section 1 presents the prologue about software testing and its inevitability, Unit testing and some brief introduction to Python. Section 2 consists of testing Python programs using *__name__* attribute and *doctest* module. Section 3 presents the details about the unit test framework of Python, i. e., *unittest* and testing Python programs using *unittest* module with examples and results.

1.1 Software Testing: The quality of a software product is the most significant aspect that the customer will be expecting implicitly. To ensure the quality of the software, it must be fussily tested. Software Testing is defined as *'the process of executing a program with the intent of finding errors'*. Thus testing always attempts to show the existence of errors but never their nonexistence. Testing makes sure our code works appropriately under a set of stated conditions. Software testing is carried out in different levels: Unit testing, Integration testing, Function testing, System testing, Acceptance testing and Installation testing.

1.2 Unit Testing: Unit testing, specifically tests a single "unit" of code in seclusion. A unit could be an entire module, a single class or function, or almost anything in between. Unit testing is very decisive in finding errors in individual programs before assimilating them into a system. Many programming languages have been using their own unit test frameworks such as JUnit (Java), PHPUnit (PHP), NUnit (.Net), CppTest and CppUnit (C++), unittest (Python).

1.3 Brief introduction to Python: Python is a high-level, interpreted, interactive and object-oriented, open source scripting language created by Guido Van Rossum. On *Angel List* (a U.S. website for startups, job seekers & startup investors), Python is the 2nd most demanded skill and also the skill with the highest average salary offered. With the rise of *Big Data*, Python developers are in demand as data scientists, especially since Python can be easily integrated into web applications to carry out tasks that require machine learning. According to the *TIOBE* programming community index (An indicator of popularity of programming languages), python is the 4th most popular programming language out of 100.

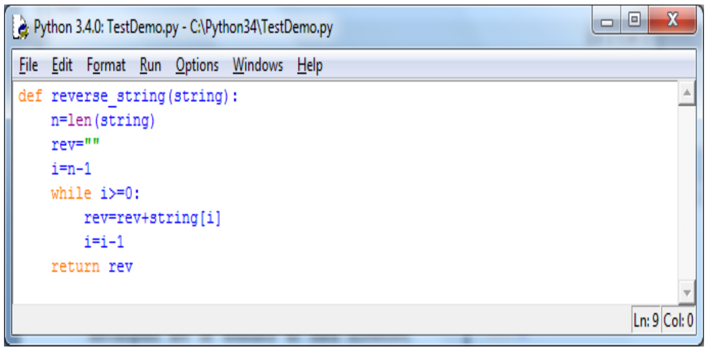
2. Testing Python Programs

Testing python programs can be done in variety of ways. We can use the popular unit testing API of Python, the '*unittest*' or we can use the '*doctest*' module or we can simply test the program using '*__name__*' attribute. In this section, we perceive how to test a Python program using '*__name__*' attribute and '*doctest*' module.

2.1 Using '*__name__*' attribute:

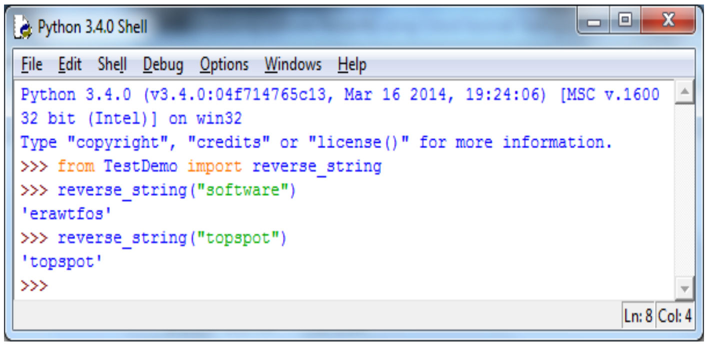
Every python module has a name, which is defined in the built-in attribute called '*__name__*'. When we execute a module as a standalone program, (eg. `$python3 abc.py`) the attribute *__name__* will be assigned the string '*__main__*'. If we imported the module `abc.py` in some other program then the *__name__* attribute possesses the value '`abc`'.

The following module `TestDemo.py` has a function '*reverse_string()*' that returns the reverse of a given string:


Python 3.4.0: TestDemo.py - C:\Python34\TestDemo.py
File Edit Format Run Options Windows Help
Ln: 9 Col: 0

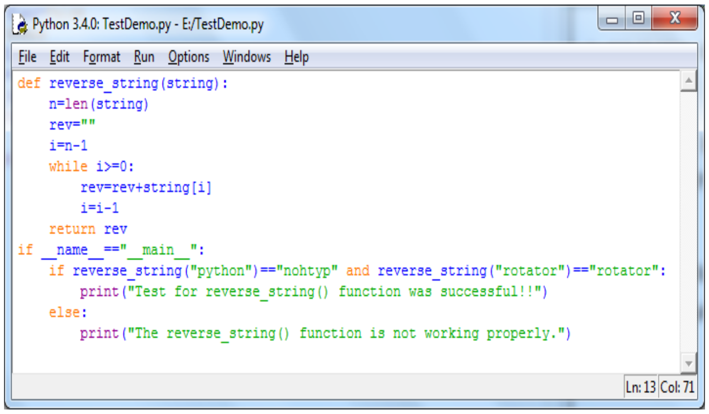
```
def reverse_string(string):
    n=len(string)
    rev=""
    i=n-1
    while i>=0:
        rev=rev+string[i]
        i=i-1
    return rev
```

We can test this module manually from the interactive shell as shown below:



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> from TestDemo import reverse_string
>>> reverse_string("software")
'erawtfos'
>>> reverse_string("topspot")
'topspot'
>>>
```

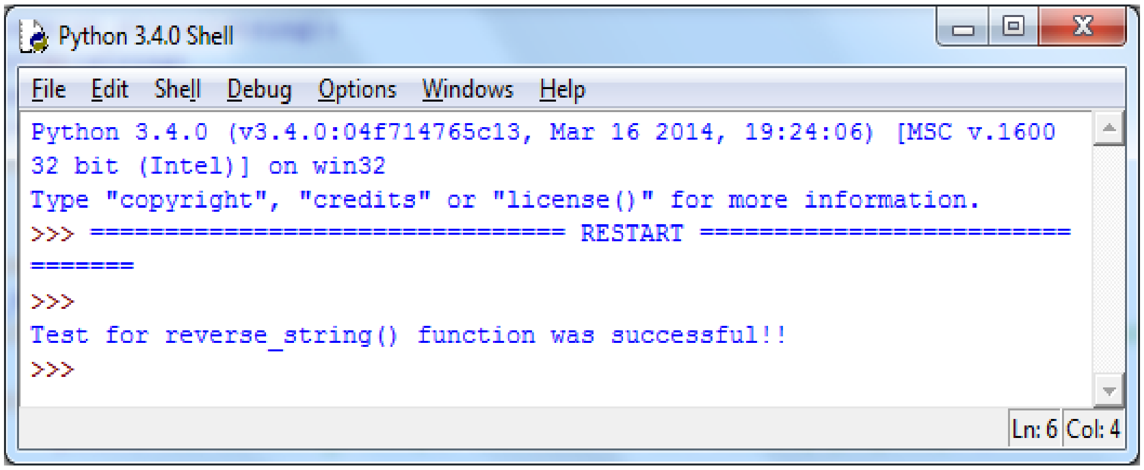
I included some piece of code after the function definition in `TestDemo.py` as shown below:


Python 3.4.0: TestDemo.py - E:/TestDemo.py
File Edit Format Run Options Windows Help
Ln: 13 Col: 71

```python
def reverse_string(string):
    n=len(string)
    rev=""
    i=n-1
    while i>=0:
        rev=rev+string[i]
        i=i-1
    return rev
if __name__=="__main__":
    if reverse_string("python")=="nohtyp" and reverse_string("rotator")=="rotator":
        print("Test for reverse_string() function was successful!!")
    else:
        print("The reverse_string() function is not working properly.")
```

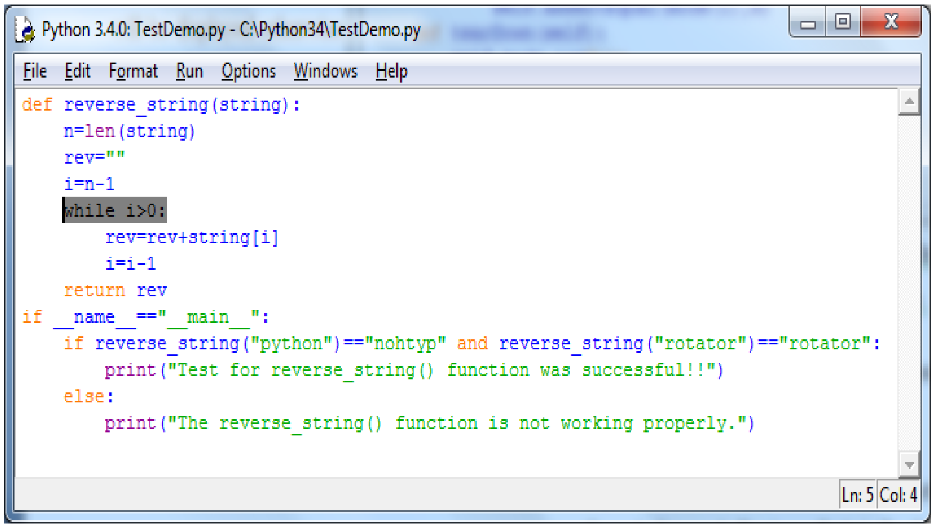
The code I added means that if the name of the module that is being run currently is '*__main__*' then test the function *reverse_string()* for some sample input. If the values returned by the function are matched with the expected values then a message indicating the success of the test will be displayed otherwise a message indicating failure of the test will be displayed.

When we run the module `TestDemo.py` (by pressing F5), we get the output as given below:



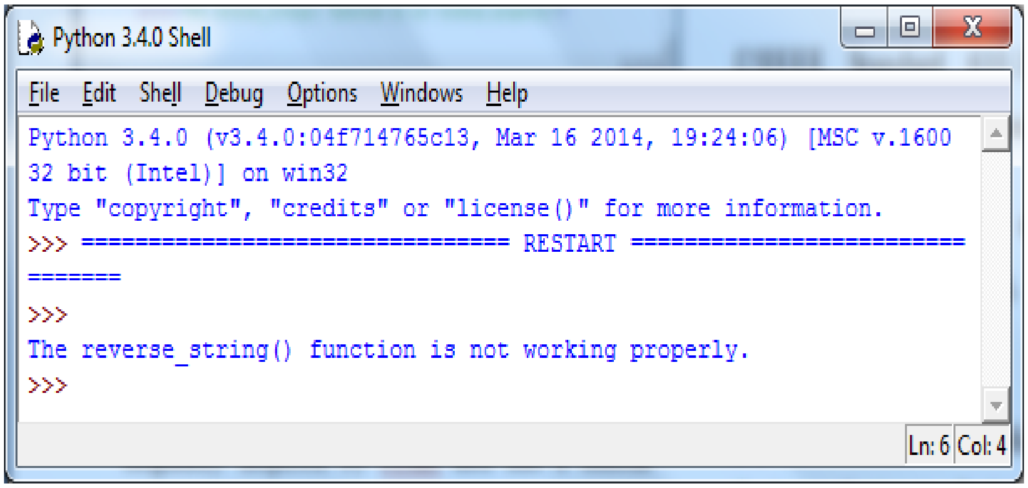
```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART =========================
=======
>>>
Test for reverse_string() function was successful!!
>>>
```

I modified the code inside the function so as to make the test to fail as shown below:


Python 3.4.0: TestDemo.py - C:\Python34\TestDemo.py
File Edit Format Run Options Windows Help
Ln: 5 Col: 4

```
def reverse_string(string):
    n=len(string)
    rev=""
    i=n-1
    while i>0:
        rev=rev+string[i]
        i=i-1
    return rev
if __name__=="__main__":
    if reverse_string("python")=="nohtyp" and reverse_string("rotator")=="rotator":
        print("Test for reverse_string() function was successful!!")
    else:
        print("The reverse_string() function is not working properly.")
```

When the above changed version of the module is executed, I got the following output:



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART =========================
=======
>>>
The reverse_string() function is not working properly.
>>>
```

This is the simplest method for unit testing python programs.

2.2 Using doctest module:

The *doctest* is a unit test framework that arrives prepackaged with python. The *doctest* module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to bear out that they work exactly as shown. There are numerous common ways to use *doctest*:

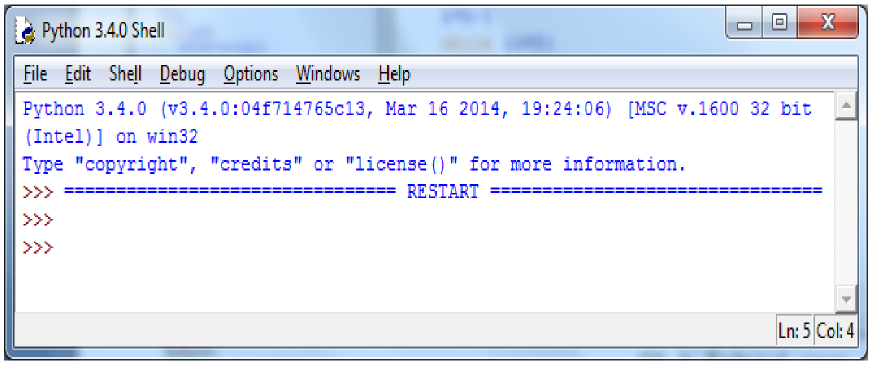
- To check that a module's docstrings are up-to-date by corroborating that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as anticipated.
- To write tutorial documentation for a package, liberally exemplified with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

To use *doctest* for unit testing, we must place a *docstring*, which looks like the output we get in the interactive shell, as given below:

```
>>>reverse_string("hello")
'olleh'
>>>reverse_string("radar")
'radar'
```

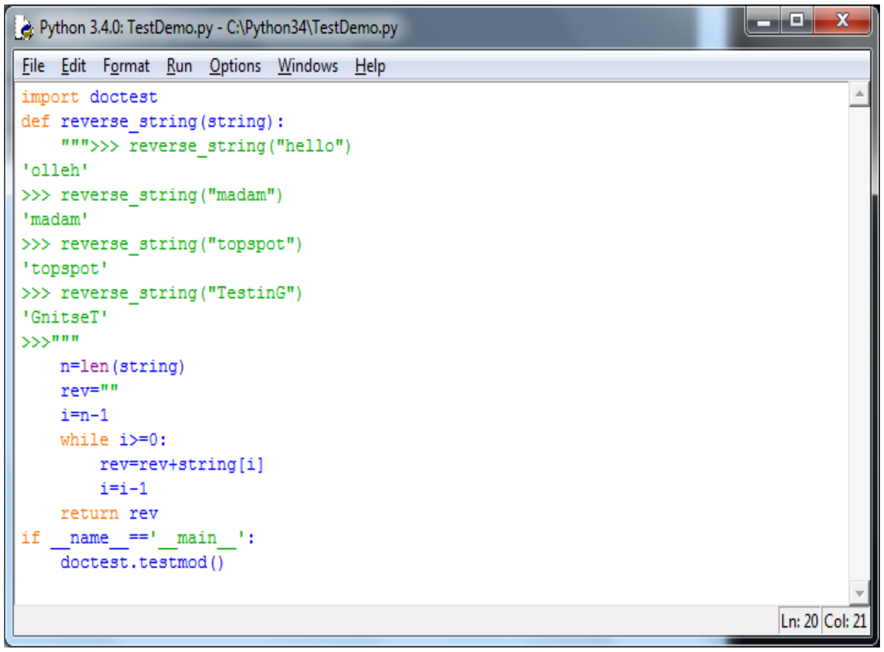
Then we must call the *testmod()* function of *doctest* to test the module. If the program is functioning as estimated (based on the *docstring*) then Python will not generate any output. Otherwise it shows the failed tests. The following is an example Python source code which demonstrates the use of *doctest*:

When we run this program in the Python IDLE (Integrated Development & Learning Environment), it produces no output as shown below:



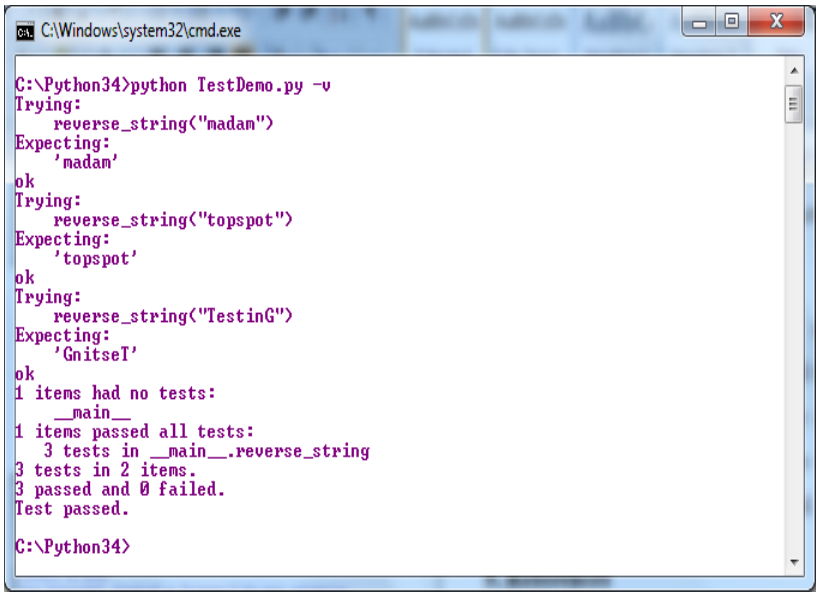
```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
>>>
```

But if we run the identical code using operating system's command prompt by passing −v as argument then it shows


Python 3.4.0: TestDemo.py - C:\Python34\TestDemo.py
File Edit Format Run Options Windows Help
Ln: 20 Col: 21

```
import doctest
def reverse_string(string):
    """>>> reverse_string("hello")
'olleh'
>>> reverse_string("madam")
'madam'
>>> reverse_string("topspot")
'topspot'
>>> reverse_string("TestinG")
'GnitseT'
>>>"""
    n=len(string)
    rev=""
    i=n-1
    while i>=0:
        rev=rev+string[i]
        i=i-1
    return rev
if __name__=='__main__':
    doctest.testmod()
```

output even if the tests are passed:

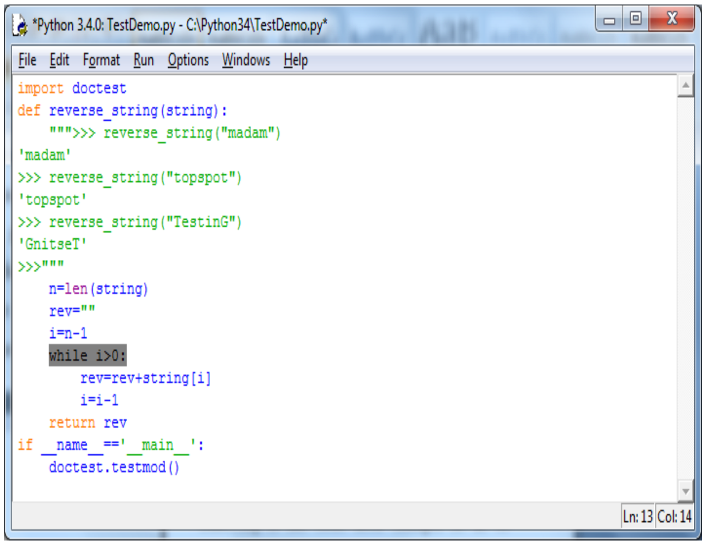


C:\Windows\system32\cmd.exe

```
C:\Python34>python TestDemo.py -v
Trying:
    reverse_string("madam")
Expecting:
    'madam'
ok
Trying:
    reverse_string("topspot")
Expecting:
    'topspot'
ok
Trying:
    reverse_string("TestinG")
Expecting:
    'GnitseT'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.reverse_string
3 tests in 2 items.
3 passed and 0 failed.
Test passed.

C:\Python34>
```

When the test fails, Python produces the output indicating expected and observed results.

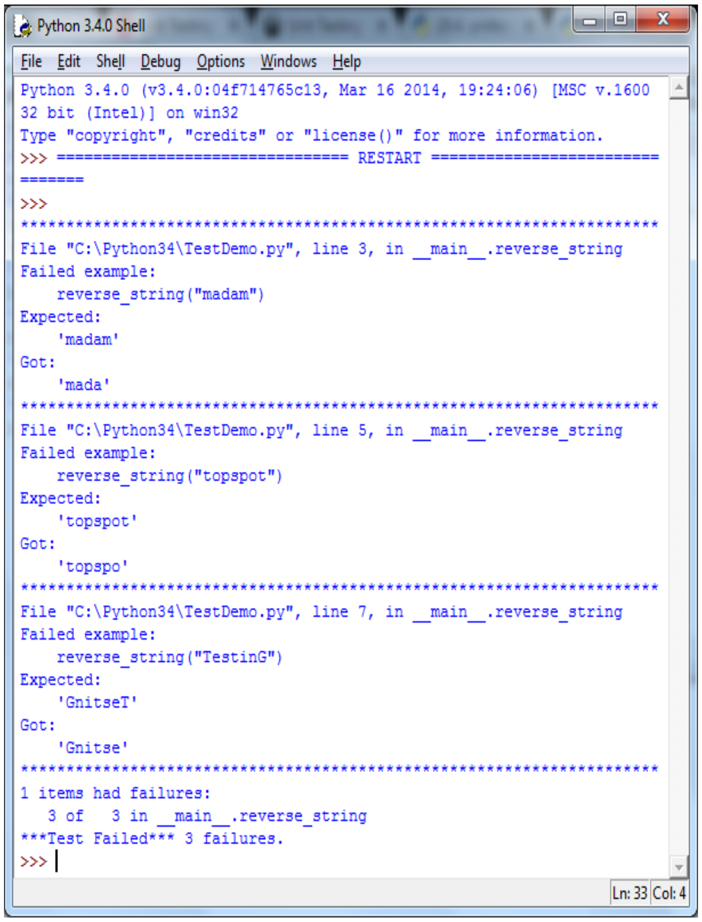
The following is the faulty code after some changes:



Python 3.4.0: TestDemo.py - C:\Python34\TestDemo.py

```
import doctest
def reverse_string(string):
    """>>> reverse_string("madam")
'madam'
>>> reverse_string("topspot")
'topspot'
>>> reverse_string("TestinG")
'GnitseT'
>>>"""
    n=len(string)
    rev=""
    i=n-1
    while i>0:
        rev=rev+string[i]
        i=i-1
    return rev
if __name__=='__main__':
    doctest.testmod()
```

When the above program is executed, it gives the following output:



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> =============================== RESTART =========================
=======
>>>
**********************************************************************
File "C:\Python34\TestDemo.py", line 3, in __main__.reverse_string
Failed example:
    reverse_string("madam")
Expected:
    'madam'
Got:
    'mada'
**********************************************************************
File "C:\Python34\TestDemo.py", line 5, in __main__.reverse_string
Failed example:
    reverse_string("topspot")
Expected:
    'topspot'
Got:
    'topspo'
**********************************************************************
File "C:\Python34\TestDemo.py", line 7, in __main__.reverse_string
Failed example:
    reverse_string("TestinG")
Expected:
    'GnitseT'
Got:
    'Gnitse'
**********************************************************************
1 items had failures:
   3 of   3 in __main__.reverse_string
***Test Failed*** 3 failures.
>>> |
```

3. *unit test*: Unit Test Framework of Python

The unit test unit testing framework was originally motivated by *JUnit* and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. This framework is appropriate for testing complex Python programs.

The subsequent terminology must be known before working with *unit test*:

- **Test fixture:** A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions.
- **Test case:** A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. *unittest* module offers a base class, *TestCase*, which may be used to create new test cases.
- **Test suite:** A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed collectively.
- **Test runner:** A *test runner* is a component which synchronizes the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

A test case is created by sub classing *unittest. Test Case*. Inside that subclass we can have as many tests as we want. The crux of each test is a call to one of the methods of *unit test. Test Case* class. The *assert Equal()* to check for an expected result; *assert True()* or *assert False()* to verify a condition; or *assert Raises()* to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

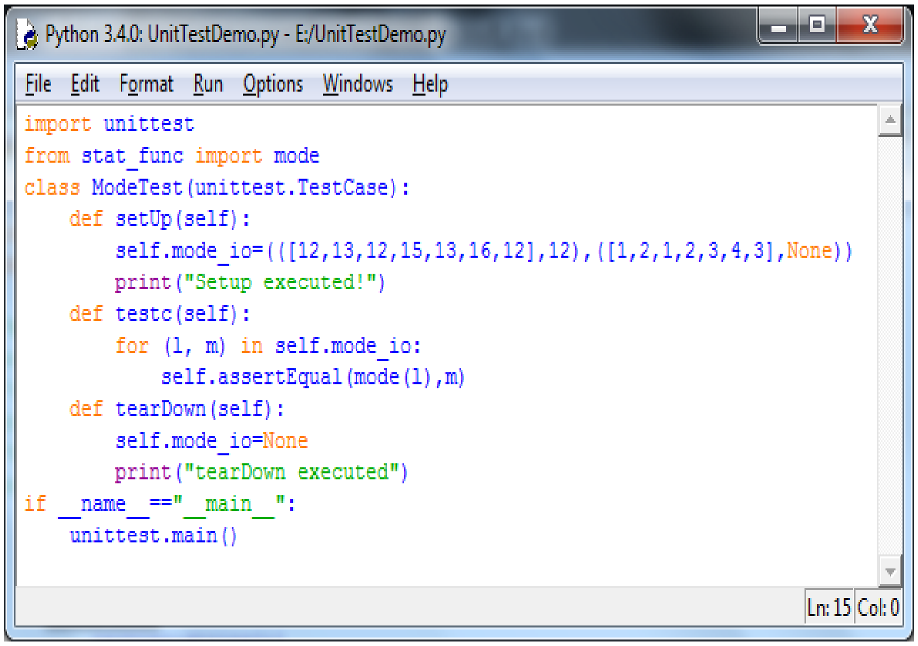
The *setUp()* and *tearDown()* methods allow you to define instructions that will be executed before and after each test method.

To write a simple test cases using *unit test* module, we need to follow the steps given below:

1. Import *unit test* module
2. Import a Python function or program on which you want to perform unit test.
3. Define a sub class of *unittest .TestCase* class.
4. Define the *setUp()* and *tearDown()* methods along with the test methods using which you want to unit test the python program.

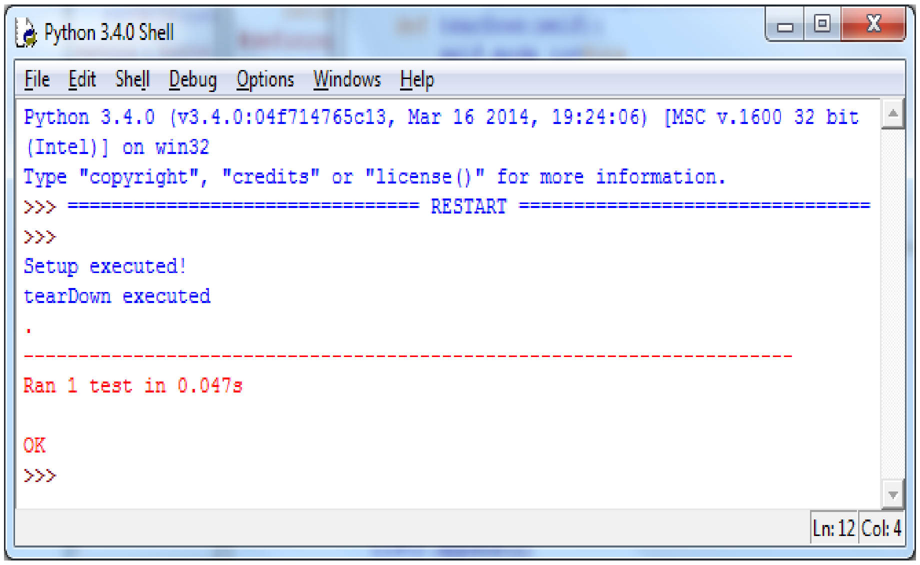
To better understand writing and running test cases, consider a Python module named 'stat_func.py' which contains three functions *mean()*, *median()* and *mode()*:

Now I write a test case to test the working of the function *mode()* of stat_func.py module as given below: (UnitTestDemo.py)


Python 3.4.0: UnitTestDemo.py - E:/UnitTestDemo.py
File Edit Format Run Options Windows Help
Ln: 15 Col: 0

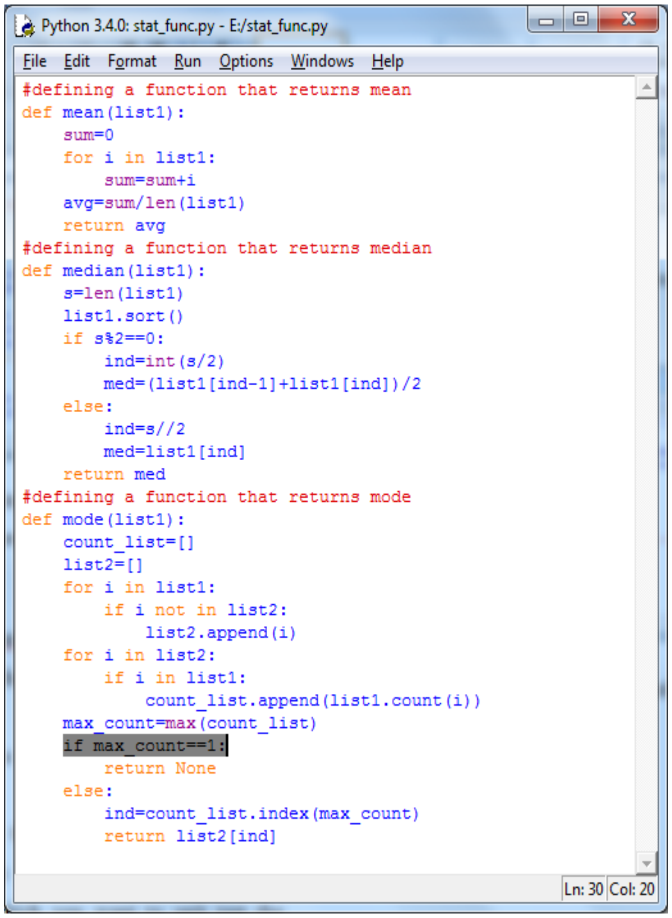
```
import unittest
from stat_func import mode
class ModeTest(unittest.TestCase):
    def setUp(self):
        self.mode_io=(([12,13,12,15,13,16,12],12),([1,2,1,2,3,4,3],None))
        print("Setup executed!")
    def testc(self):
        for (l, m) in self.mode_io:
            self.assertEqual(mode(l),m)
    def tearDown(self):
        self.mode_io=None
        print("tearDown executed")
if __name__=="__main__":
    unittest.main()
```

When we run the above program, we get the output as shown below:



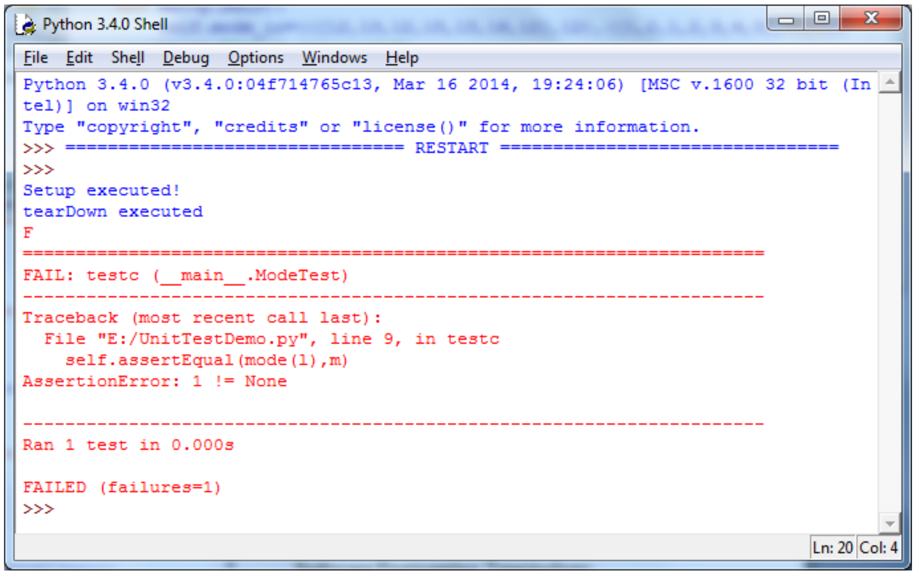
```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Setup executed!
tearDown executed
.
----------------------------------------------------------------------
Ran 1 test in 0.047s

OK
>>>
```

Here 'OK' indicates successful execution of the test cases.
If any one of the test cases is failed then Python indicates the failure by showing an *AssertionError*.
I intentionally changed a line of code inside the function so that the test case fails:


Python 3.4.0: stat_func.py - E:/stat_func.py
File Edit Format Run Options Windows Help
Ln: 30 Col: 20

```
#defining a function that returns mean
def mean(list1):
    sum=0
    for i in list1:
        sum=sum+i
    avg=sum/len(list1)
    return avg
#defining a function that returns median
def median(list1):
    s=len(list1)
    list1.sort()
    if s%2==0:
        ind=int(s/2)
        med=(list1[ind-1]+list1[ind])/2
    else:
        ind=s//2
        med=list1[ind]
    return med
#defining a function that returns mode
def mode(list1):
    count_list=[]
    list2=[]
    for i in list1:
        if i not in list2:
            list2.append(i)
    for i in list2:
        if i in list1:
            count_list.append(list1.count(i))
    max_count=max(count_list)
    if max_count==1:
        return None
    else:
        ind=count_list.index(max_count)
        return list2[ind]
```

When I ran the `UnitTestDemo.py` module after the above change in the *mode()* function, the output shown below occurred:



```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Setup executed!
tearDown executed
F
======================================================================
FAIL: testc (__main__.ModeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "E:/UnitTestDemo.py", line 9, in testc
    self.assertEqual(mode(1),m)
AssertionError: 1 != None

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
>>>
```

The letter ‘F’ indicates the failure of the test case.

4. Conclusion:

In this paper, I presented various approaches available for unit testing Python programs. Using simple Python modules, I demonstrated how these approaches are used. I also presented the outputs when those tests are performed on the Python programs. Along with the methods I presented here, there are some other ways to perform unit testing in Python such as using *PyTest* module, using nose, using mock etc. I conclude that Python offers easy-to-use and useful mechanisms for the Unit Testing of programs.

5. References:

1. “Python Essential Reference”, 3rd edition, David M. Beazley, Sams Publishing, pp: 503-516.
2. “Python Essential Reference”, 4th edition, David M. Beazley, Pearson Education, pp: 181-186.
3. https://docs.python.org.
4. “The Hitchhiker’s Guide to Python-*Best Practices for Development*”, Kenneth Reitz and Tanya Schlusser, O’Reilly.
5. “Programming Python”, 4th edition, Mark Lutz, O’reilly.