Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #90
Testing - pytest and doctest

By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M(Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Introduction to Frameworks

One of the most popular and effective programming language that contains **vast libraries and frameworks for almost every technical domain is PYTHON**.

Python frameworks automate the implementation of several tasks and give developers a structure for application development. Key feature is **flexibility - They provide the necessary tools and features that developers can extend and build upon.** Each framework comes with its own **collection of modules or packages that significantly reduce development time**. A Python framework can either be **full-stack, micro, or asynchronous.**

- **A full-stack framework** contains all the web development requirements. These include form generators, template layouts, form validation, and lots more.
- **Micro framework** requires a bit of manual work from developers. It is used for minimalistic web application development
- **Asynchronous framework** uses the asyncio library to run a number of processes concurrently. They can handle a large number of concurrent connections.

Benefits of Frameworks

- Easier implementation and easy integration
- Efficient operation and good documentation
- Code reusability, flexible and secure
- Open-source

Let us deal with **testing frameworks in python**.

What is software testing? - A process by which one or more expected behaviors and results from a piece of software are exercised and confirmed. Testing can be done either manually or automation. Few of the available python testing frameworks are – DJango, Web2Py, Flask, Bottle, CherryPy, Robot, PyTest, Nose2, Testif, etc.

Let us discuss Pytest framework in detail.

Framework - pytest

A Python testing framework that originated from the PyPy project. It can be used to write various types of software tests, including unit tests**, integration tests, end-to-end tests, and functional tests**. Test cases can be included using any of these ways - **parametrized testing, fixtures, and assert re-writing.**

Installation steps:

Step1: Installation in command line

pip install pytest

or

pip install -U pytest

Step 2: Installation confirmation

py.test –h

or

pytest –version

Features:

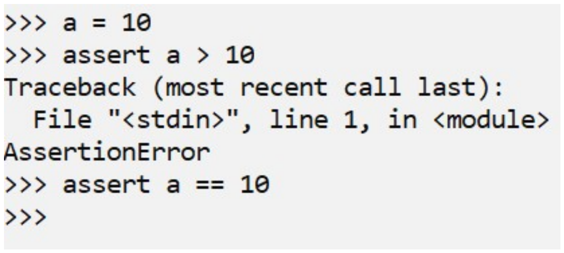
- Provides several **options to run tests from the command line.**
- Very easy to get started with **creating test cases.**
- **Allows multiple tests to run in parallel**, which reduces the execution time.
- **Automatically detects the test file and test functions**, if not mentioned explicitly.
- Allows us to **skip the tests during execution** and allows running a subset of the entire test suite.

Let us understand the **assert statements** before jumping to assert rewriting method of writing the test cases. Assert statement **allows you to write sanity checks in your code.** These checks are known as **assertions**, and you can **use them to test if certain assumptions remain True while you are developing your code**. If any of your assertions turn False, then you have a bug in your code.

Syntax:

assert expression ["," assertion_message]

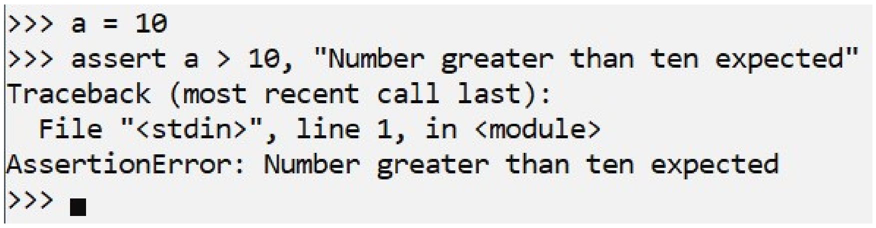
Here, expression can be any valid expression which is then tested for truthiness. If expression is False, then the statement throws an AssertionError. The assertion_message parameter is optional but encouraged. It can hold a string describing the issue that the statement is supposed to catch.



```
>>> a = 10
>>> assert a > 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert a == 10
>>>
```

With a truthy expression, the assertion succeeds, and nothing happens. In that case, your program continues its normal execution. In contrast, a falsy expression makes the assertion fail, raising an AssertionError and breaking the program's execution.

To make your assert statements clear to other developers, better to add a descriptive assertion message.



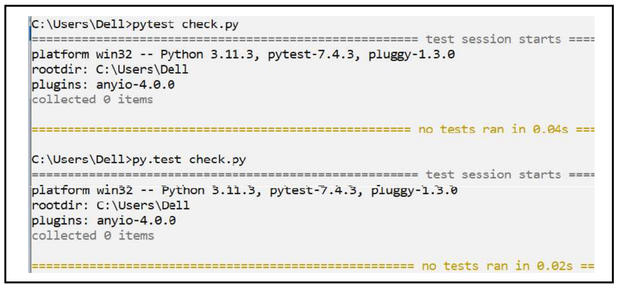
```
>>> a = 10
>>> assert a > 10, "Number greater than ten expected"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Number greater than ten expected
>>> ■
```

Can we put the expression and assertion_message in a tuple? Try this!

Now, let us use this in our test case by applying assert rewriting method of Pytest.

Example_code_1: check.py

```
def f1():
   n = 9
   assert n == 9, n
   print("after assert")
import math
def mysqrt():
   assert math.sqrt(26) == 5
   print("done")
```



```
C:\Users\Dell>pytest check.py
==================================================== test session starts ====
platform win32 -- Python 3.11.3, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Users\Dell
plugins: anyio-4.0.0
collected 0 items

=================================================== no tests ran in 0.04s ===

C:\Users\Dell>py.test check.py
==================================================== test session starts ====
platform win32 -- Python 3.11.3, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Users\Dell
plugins: anyio-4.0.0
collected 0 items

==================================================== no tests ran in 0.02s ==
```

In the above code, we did not follow any rules either for the file containing the test cases or any of the test cases / functions within the file. Let us note these rules and then continue coding.

- By default, **pytest only identifies the file names starting with test_** or ending with **_test.py** as the test files in the current directory and its subdirectories.

- **Test case or function has to start with "test"** word followed by any number of characters.

Note: It is not possible to make pytest to consider any function not starting with "test" as the test case.

Few Examples of valid and invalid pytest file names.

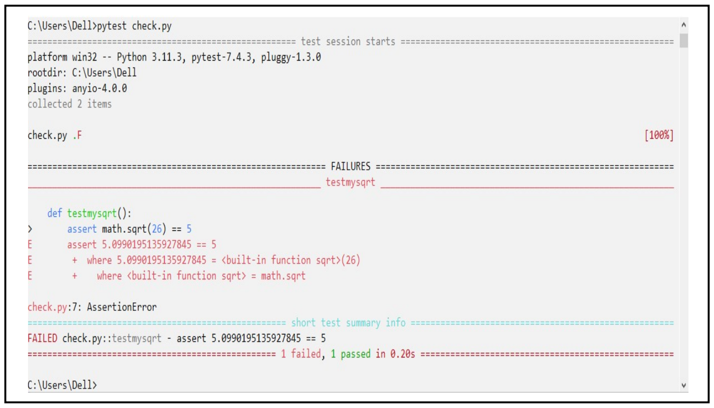
- test_one.py - valid
- one_test.py - valid
- testtwo.py -invalid
- twotest.py -invalid

Few Examples of valid and invalid pytest test cases/methods

- def test_method(): - valid
- def testmethod(): - valid
- def method_test(): - invalid

Example_code_2: check.py. File name not modified. Function names modified

```
def test_f1():
   n = 9
   assert n == 9, n
   print("after assert")
import math
def testmysqrt():
   assert math.sqrt(26) == 5
   print("done")
```



```
C:\Users\Dell>pytest check.py
======================================================= test session starts =======================================================
platform win32 -- Python 3.11.3, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Users\Dell
plugins: anyio-4.0.0
collected 2 items

check.py .F                                                                                                                  [100%]

============================================================ FAILURES =============================================================
___________________________________________________________ testmysqrt ____________________________________________________________

    def testmysqrt():
>       assert math.sqrt(26) == 5
E       assert 5.0990195135927845 == 5
E        +  where 5.0990195135927845 = <built-in function sqrt>(26)
E        +    where <built-in function sqrt> = math.sqrt

check.py:7: AssertionError
===================================================== short test summary info =====================================================
FAILED check.py::testmysqrt - assert 5.0990195135927845 == 5
=================================================== 1 failed, 1 passed in 0.20s ===================================================

C:\Users\Dell>
```

Note: The file containing the test cases is given to the pytest command directly. Hence name of the file doesn't matter. Pytest automatically detects the test cases and runs it and reports it. Observe .F in the output.

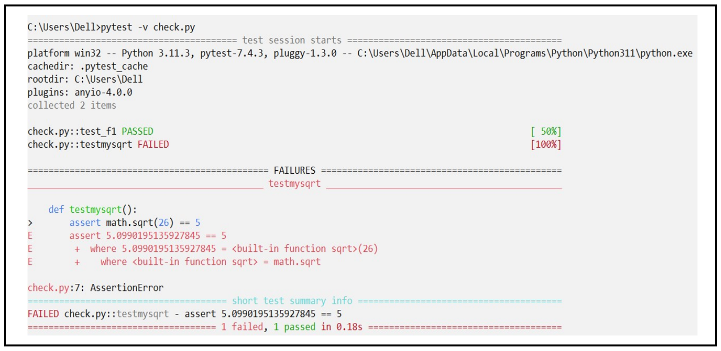
Dot(.) means successful test case and F means Failed test case. Also this contains the details of failed test cases only.

How to **get the details of successful test cases along with the report of failures? Try verbose option (-v)**

pytest -v check.py

Example_code_3: One more function/test case added to check.py

```
def test_f1():
    n = 9
    assert n == 9, n
    print("after assert")
import math
def testmysqrt():
    assert math.sqrt(26) == 5
    print("done")
def decrementtest():
    n = 10
    assert 9 == n-1, "decremented value compared"
    print("after assert in decrementtest")
```



```
C:\Users\Dell>pytest -v check.py
======================================= test session starts ========================================
platform win32 -- Python 3.11.3, pytest-7.4.3, pluggy-1.3.0 -- C:\Users\Dell\AppData\Local\Programs\Python\Python311\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Dell
plugins: anyio-4.0.0
collected 2 items

check.py::test_f1 PASSED                                                                    [ 50%]
check.py::testmysqrt FAILED                                                                 [100%]

============================================= FAILURES =============================================
____________________________________________ testmysqrt ____________________________________________

    def testmysqrt():
>       assert math.sqrt(26) == 5
E       assert 5.0990195135927845 == 5
E        +  where 5.0990195135927845 = <built-in function sqrt>(26)
E        +    where <built-in function sqrt> = math.sqrt

check.py:7: AssertionError
===================================== short test summary info ======================================
FAILED check.py::testmysqrt - assert 5.0990195135927845 == 5
=================================== 1 failed, 1 passed in 0.18s ====================================
```

Note: pytest doesn’t recognize the added test case as the naming convention is not followed for this test case. Also note **PASSED** and **FAILED** words in the output as –v is used.

Example_code_4: To run more than one but not all the files in the folder use

Create a folder. In our example, automation. Inside this, create three test files as below. Run only 2 out of it.

py.test <specify files separated by a space in between >

test_example.py

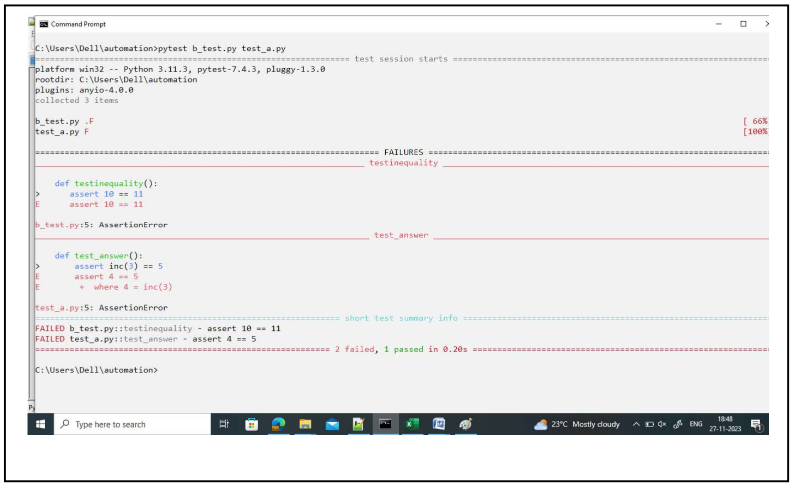
```
import math
def sqrt_test():
      num = 25
      assert math.sqrt(num) != 5
def testsquare():
        num = 7
        assert 7*7 == 40
def testequality():
              assert 10 == 11
```

test_a.py

```
def inc(x):
   return x + 1

def test_answer():
   assert inc(3) == 5
```

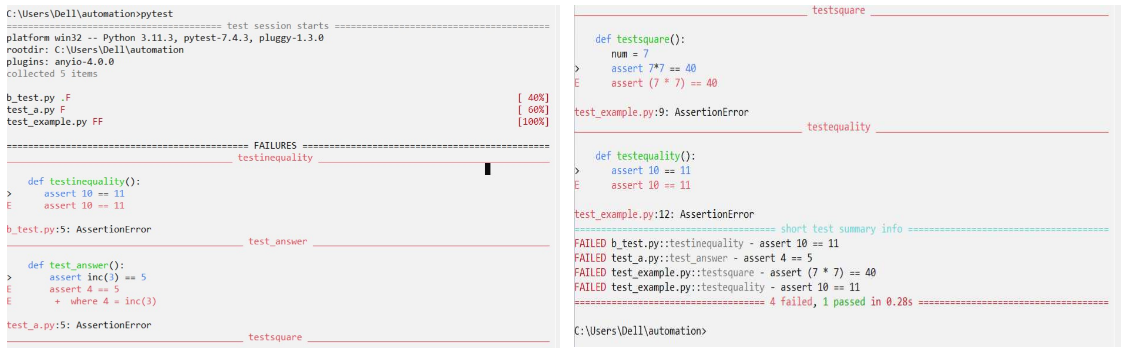
b_test.py

```
def testcube():
  num = 5
  assert num*num*num == 125

def testinequality():
  assert 10 == 11
```



```
C:\Users\Dell\automation>pytest b_test.py test_a.py
=============================================================== test session starts ===============================================================
platform win32 -- Python 3.11.3, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Users\Dell\automation
plugins: anyio-4.0.0
collected 3 items

b_test.py .F                                                                                                                                  [ 66%]
test_a.py F                                                                                                                                   [100%]

==================================================================== FAILURES =====================================================================
_________________________________________________________________ testinequality __________________________________________________________________

    def testinequality():
>       assert 10 == 11
E       assert 10 == 11

b_test.py:5: AssertionError
__________________________________________________________________ test_answer ____________________________________________________________________

    def test_answer():
>        assert inc(3) == 5
E        assert 4 == 5
E         +  where 4 = inc(3)

test_a.py:5: AssertionError
============================================================= short test summary info =============================================================
FAILED b_test.py::testinequality - assert 10 == 11
FAILED test_a.py::test_answer - assert 4 == 5
=========================================================== 2 failed, 1 passed in 0.20s ===========================================================

C:\Users\Dell\automation>
```

Example_code_5: In the above example, if you want the Pytest to automatically detect the test cases, just say Pytest or py.test and press enter.



```
C:\Users\Dell\automation>pytest
======================================== test session starts ========================================
platform win32 -- Python 3.11.3, pytest-7.4.3, pluggy-1.3.0
rootdir: C:\Users\Dell\automation
plugins: anyio-4.0.0
collected 5 items

b_test.py .F                                                                                  [ 40%]
test_a.py F                                                                                   [ 60%]
test_example.py FF                                                                            [100%]

============================================= FAILURES ==============================================
__________________________________________ testinequality ___________________________________________

    def testinequality():
>       assert 10 == 11
E       assert 10 == 11

b_test.py:5: AssertionError
____________________________________________ test_answer ____________________________________________

    def test_answer():
>        assert inc(3) == 5
E        assert 4 == 5
E         +  where 4 = inc(3)

test_a.py:5: AssertionError
____________________________________________ testsquare _____________________________________________
```

```
____________________________________________ testsquare _____________________________________________

    def testsquare():
        num = 7
>       assert 7*7 == 40
E       assert (7 * 7) == 40

test_example.py:9: AssertionError
___________________________________________ testequality ____________________________________________

    def testequality():
>       assert 10 == 11
E       assert 10 == 11

test_example.py:12: AssertionError
===================================== short test summary info =====================================
FAILED b_test.py::testinequality - assert 10 == 11
FAILED test_a.py::test_answer - assert 4 == 5
FAILED test_example.py::testsquare - assert (7 * 7) == 40
FAILED test_example.py::testequality - assert 10 == 11
==================================== 4 failed, 1 passed in 0.28s ====================================

C:\Users\Dell\automation>
```

Think about this:

Since all the test cases are starting with test word, how do you choose a subset of test cases for testing by Pytest framework. Is there any option available? Ans: -k

Try this!

-- End of Pytest –

Module, a lightweight testing framework – doctest

This provides a quick and straightforward test automation. It can read the test cases from project's documentation and code's docstrings. It **searches for test cases in your documentation and the docstrings of packages, modules, functions, classes, and methods**. It **doesn't search for test cases in any objects that you import**.

To find and run your test cases, doctest follows a few steps:

1. Searches for text that looks like Python interactive sessions in the documentation and docstrings
2. Parses those pieces of text to distinguish between executable code and expected results
3. Runs the executable code like regular Python code
4. Compares the execution result with the expected result

In general, doctest interprets all those lines of text that start with the primary (>>>) or secondary (...) REPL prompts. The lines immediately after either prompt are understood as the code's expected output or result.

There are several purposes where doctest can be used.

- Writing **quick and effective test cases** to check your code as you write it
- Running **acceptance**, **regression** and **integration** test cases on your projects, packages, and modules – [DO NOT WORRY what are these as of now.]
- Checking if **docstrings** are **up-to-date** and in **sync** with the target code
- Verifying if projects' **documentation** is **up-to-date**
- Writing **hands-on tutorials** for projects, packages, and modules
- Illustrating how to **use projects' APIs** and what the expected input and output must be.

The functions **testmod()** and **testfile()** provide a simple interface to doctest that should be sufficient for most basic uses.

Let us try understanding the working of it with an example code.

```
def fact(n):
   if (n == 0): return 1
   else:
      return n*fact(n-1)
```

The above function we want to test with multiple inputs. How do we do this usually? We call the function with different inputs as below. Comment and uncomment the function calls to see the output of all.

```
print(fact(5))
print(fact(1))
print(fact(0))
print(fact(-1))
print(fact(30))
print(fact(9.3))
print(fact(4))
print(fact(-5))
```

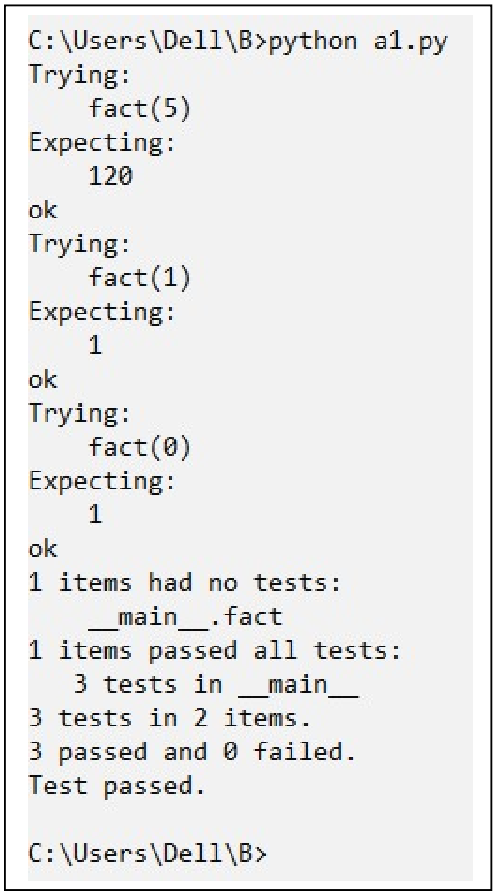
But if the output is an Exception, none of the below statement will be executed. This is the problem if we test it this way. As a solution to this, can we add **all these with the same code in the document string**?

Example_code_1: This code doesn't output anything as these is no failed test case.

```
"""
   >>> fact(5)   #there is a space between >>> and f
   120
   >>> fact(1)
   1
   >>> fact(0)
   1
"""
import doctest
def fact(n):
   if (n == 0): return 1
   else:
       return n*fact(n-1)
doctest.testmod()
```

Example_code_2: Addition of verbose = True to the testmod() function

```
"""
    >>> fact(5)
    120
    >>> fact(1)
    1
    >>> fact(0)
    1
"""
import doctest
def fact(n):
    if (n == 0): return 1
    else:
        return n*fact(n-1)
doctest.testmod(verbose = True)
```

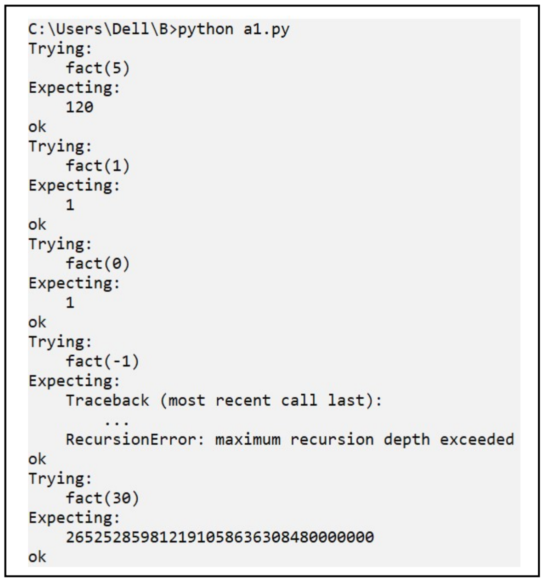


```
C:\Users\Dell\B>python a1.py
Trying:
    fact(5)
Expecting:
    120
ok
Trying:
    fact(1)
Expecting:
    1
ok
Trying:
    fact(0)
Expecting:
    1
ok
1 items had no tests:
    __main__.fact
1 items passed all tests:
   3 tests in __main__
3 tests in 2 items.
3 passed and 0 failed.
Test passed.

C:\Users\Dell\B>
```

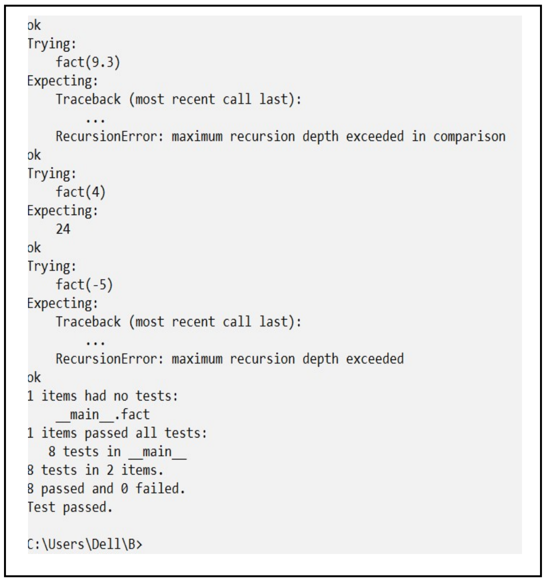
Example_code_3: Now adding a few test cases which throw some exception. verbose = True is also maintained to get the report on successful test cases too.

```
"""
    >>> fact(5)
    120
    >>> fact(1)
    1
    >>> fact(0)
    1
    >>> fact(-1)
    Traceback (most recent call last):
       ...
    RecursionError: maximum recursion depth exceeded
    >>> fact(30)
    265252859812191058636308480000000
    >>> fact(9.3)
    Traceback (most recent call last):
       ...
    RecursionError: maximum recursion depth exceeded in comparison
    >>> fact(4)
    24
    >>> fact(-5)
    Traceback (most recent call last):
       ...
    RecursionError: maximum recursion depth exceeded
"""
import doctest
def fact(n):
    if (n == 0): return 1
    else:
        return n*fact(n-1)
doctest.testmod(verbose = True)
```

Note: In the above code all test cases are passed. Observe the outputs below – 8 passed and 0 failed.



```
C:\Users\Dell\B>python a1.py
Trying:
    fact(5)
Expecting:
    120
ok
Trying:
    fact(1)
Expecting:
    1
ok
Trying:
    fact(0)
Expecting:
    1
ok
Trying:
    fact(-1)
Expecting:
    Traceback (most recent call last):
        ...
    RecursionError: maximum recursion depth exceeded
ok
Trying:
    fact(30)
Expecting:
    265252859812191058636308480000000
ok
```



```
ok
Trying:
    fact(9.3)
Expecting:
    Traceback (most recent call last):
        ...
    RecursionError: maximum recursion depth exceeded in comparison
ok
Trying:
    fact(4)
Expecting:
    24
ok
Trying:
    fact(-5)
Expecting:
    Traceback (most recent call last):
        ...
    RecursionError: maximum recursion depth exceeded
ok
1 items had no tests:
    __main__.fact
1 items passed all tests:
   8 tests in __main__
8 tests in 2 items.
8 passed and 0 failed.
Test passed.

C:\Users\Dell\B>
```

Expected output for test cases which throws exception are mentioned as Exception as described below:

Traceback (most recent call last):

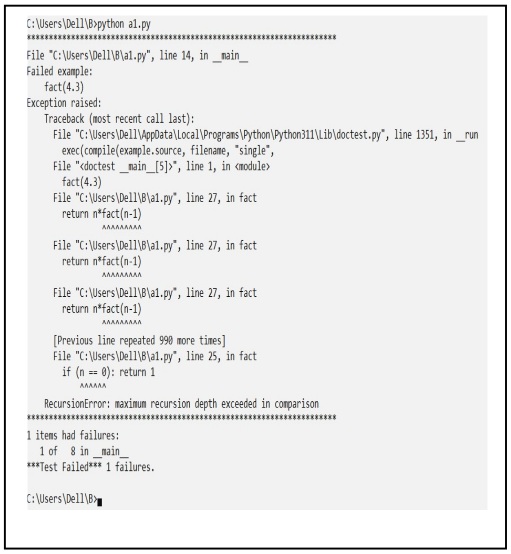
...

ExceptionName: description of exception

The actual output is matching with the expected output. Hence the test case is passed.

Example_code_4: If verbose not specified for testmod function, what is the output?

```
"""
   >>> fact(5)
   120
   >>> fact(1)
   1
   >>> fact(0)
   1
   >>> fact(-1)
   Traceback (most recent call last):
      ...
   RecursionError: maximum recursion depth exceeded
   >>> fact(30)
   265252859812191058636308480000000
   >>> fact(4.3)
   24
   >>> fact(4)
   24
   >>> fact(-5)
   Traceback (most recent call last):
      ...
   RecursionError: maximum recursion depth exceeded
"""
import doctest
def fact(n):
   if (n == 0): return 1
   else:
      return n*fact(n-1)
doctest.testmod() # verbose removed
```



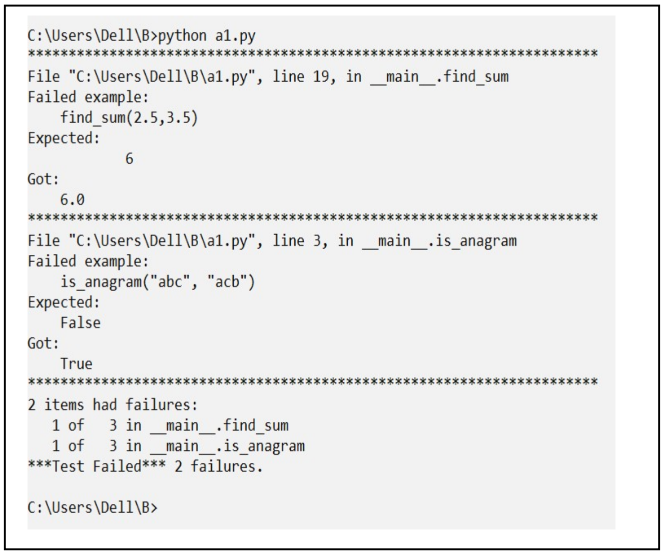
```
C:\Users\Dell\B>python a1.py
**********************************************************************
File "C:\Users\Dell\B\a1.py", line 14, in __main__
Failed example:
    fact(4.3)
Exception raised:
    Traceback (most recent call last):
      File "C:\Users\Dell\AppData\Local\Programs\Python\Python311\Lib\doctest.py", line 1351, in __run
        exec(compile(example.source, filename, "single",
      File "<doctest __main__[5]>", line 1, in <module>
        fact(4.3)
      File "C:\Users\Dell\B\a1.py", line 27, in fact
        return n*fact(n-1)
                 ^^^^^^^^^
      File "C:\Users\Dell\B\a1.py", line 27, in fact
        return n*fact(n-1)
                 ^^^^^^^^^
      File "C:\Users\Dell\B\a1.py", line 27, in fact
        return n*fact(n-1)
                 ^^^^^^^^^
      [Previous line repeated 990 more times]
      File "C:\Users\Dell\B\a1.py", line 25, in fact
        if (n == 0): return 1
            ^^^^^^
    RecursionError: maximum recursion depth exceeded in comparison
**********************************************************************
1 items had failures:
   1 of   8 in __main__
***Test Failed*** 1 failures.

C:\Users\Dell\B>
```

Few points to think!

- **In the above code, can we specify the doc string for a function rather than the python file itself?**
- **If doc string available in multiple places, what is the sequence in which it will be evaluated using testmod() function?**
- **If the file contains many functions, can we specify the function name related to which we want to run test cases?**

Example_code_5:

```
def is_anagram(a_word, b_word):
        """
        >>> is_anagram("abc", "acb")
        False
        >>> is_anagram("silent", "listen")
        True
        >>> is_anagram("one", "two")
        False
        """
        return sorted(a_word) == sorted(b_word)

def find_sum(a,b):
        """
        This function will find sum of numbers, e.g.,
        >>> find_sum(1,7)
        8
        >>> find_sum(-2,3)
        1
        >>> find_sum(2.5,3.5)
        6
        """
        return a+b
if __name__ == "__main__":
        import doctest
        doctest.testmod()
```

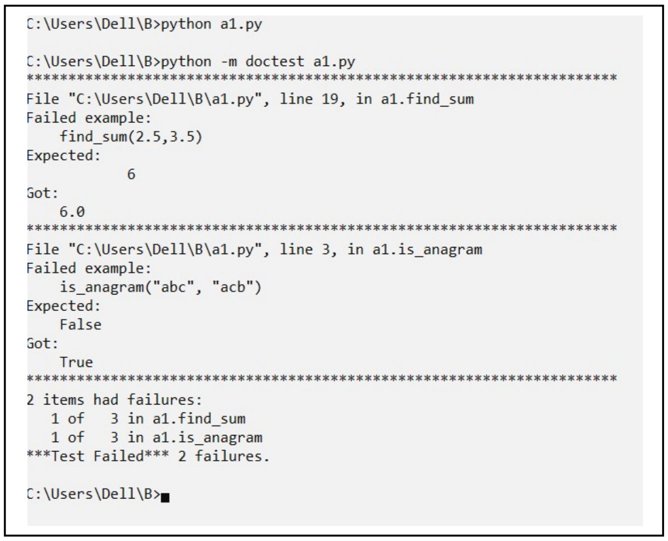


```
C:\Users\Dell\B>python a1.py
**********************************************************************
File "C:\Users\Dell\B\a1.py", line 19, in __main__.find_sum
Failed example:
    find_sum(2.5,3.5)
Expected:
            6
Got:
    6.0
**********************************************************************
File "C:\Users\Dell\B\a1.py", line 3, in __main__.is_anagram
Failed example:
    is_anagram("abc", "acb")
Expected:
    False
Got:
    True
**********************************************************************
2 items had failures:
   1 of   3 in __main__.find_sum
   1 of   3 in __main__.is_anagram
***Test Failed*** 2 failures.

C:\Users\Dell\B>
```

Example_code_6: If the python file doesn't import doctest, can you run it from outside?

Yes. Using –m option

```
def is_anagram(a_word, b_word):
        """
        >>> is_anagram("abc", "acb")
        False
        >>> is_anagram("silent", "listen")
        True
        >>> is_anagram("one", "two")
        False
        """
        return sorted(a_word) == sorted(b_word)

def find_sum(a,b):
        """
        This function will find sum of numbers, e.g.,
        >>> find_sum(1,7)
        8
        >>> find_sum(-2,3)
        1
        >>> find_sum(2.5,3.5)
        6
        """
        return a+b
```



```
C:\Users\Dell\B>python a1.py

C:\Users\Dell\B>python -m doctest a1.py
**********************************************************************
File "C:\Users\Dell\B\a1.py", line 19, in a1.find_sum
Failed example:
    find_sum(2.5,3.5)
Expected:
            6
Got:
    6.0
**********************************************************************
File "C:\Users\Dell\B\a1.py", line 3, in a1.is_anagram
Failed example:
    is_anagram("abc", "acb")
Expected:
    False
Got:
    True
**********************************************************************
2 items had failures:
   1 of   3 in a1.find_sum
   1 of   3 in a1.is_anagram
***Test Failed*** 2 failures.

C:\Users\Dell\B>
```

How do you get the report of successful test cases? Use –v with the above command.

Try this!

Add a test case to our function fact(n) where the fact function must be run on a range of values from 3 to 10 and must create a list of factorial of all numbers from 3 to 10.

-- End of Pytest –

-END-