

Exp No. 1	Write Test Case for Python Function	REG. NO:
Date:		URK23CS1197

Objective:

To develop and execute a comprehensive test case for a Python function to ensure its reliability, efficiency, and correctness.

Job Role:

Software Engineer (QA/Testing)

Skills Required:

Programming Languages: Proficiency in Python.

Testing Frameworks: Familiarity with testing frameworks like unittest, pytest, or nose.

Version Control: Knowledge of version control systems like Git.

Prerequisites:

Understanding of Software Development Life Cycle (SDLC): Familiarity with Agile methodologies.

Description:

In software engineering, a test case is a specification of the inputs, execution conditions, testing

procedure, and expected results that define a single test to be executed to achieve a particular software testing objective, such as to exercise a particular program path or to verify compliance

with a specific requirement.

It will lay out particular variables that QAs need to compare expected and actual results to conclude if the feature works. Test case components mention input, execution, and expected output/response. It tells engineers what to do, how to do it, and what results are acceptable.

The Objective of Writing Test Cases in Software Testing

To validate specific features and functions of the software.

To guide testers through their day-to-day hands-on activity.

To record a catalog of steps undertaken, which can be revisited in the event of a bug popping up.

To provide a blueprint for future projects and testers so they don't have to start work from scratch.

To help detect usability issues and design gaps early on.

To help new testers and devs quickly pick up testing, even if they join in the middle of an ongoing project.

Commands used:

```
# Run tests with unittest
python -m unittest test_file.py

# Run tests with pytest
pytest test_file.py # (requires pytest to be installed)

# Install pytest if not available
pip install pytest
```

Questions:

- 1. Design and evaluate a test case for a Python function that calculates the factorial of a given number, ensuring comprehensive coverage of both positive and negative input values, including zero.**

Program:

```
def factorial(a):
    if a==0:
        return 0
    elif a==1:
        return 1
    elif a<0:
        return "negative"
    else:
        return a * factorial(a-1)
```

TestCase:

```
import unittest
from fibo import fibonacci_upto
class TestFibonacciFunction(unittest.TestCase):

    def test_positive_input(self):
        self.assertEqual(fibonacci_upto(1), [0, 1])
        self.assertEqual(fibonacci_upto(5), [0, 1, 1, 2, 3, 5])
        self.assertEqual(fibonacci_upto(10), [0, 1, 1, 2, 3, 5, 8])

    def test_zero_input(self):
        self.assertEqual(fibonacci_upto(0), [])

    def test_one_input(self):
        self.assertEqual(fibonacci_upto(1), [0, 1])

    def test_negative_input(self):
        with self.assertRaises(ValueError):
            fibonacci_upto(-5)
```

```

def test_large_input(self):
    result = fibonacci_upto(100)
    expected = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
    self.assertEqual(result, expected)

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

```

Expected Output:

```

=====
platform win32 -- Python 3.11.9, pytest-8.4.1, pluggy-1.6.0
rootdir: C:\Users\niran\Desktop\ML OPS\EX_!\q1
plugins: mock-3.14.1
collected 4 items

test_fact.py .... [100%]

===== 4 passed in 0.11s =====
PS C:\Users\niran\Desktop\ML OPS\EX_!\q1> 

```

2. Develop and justify a test case for a Python function that checks whether a given string is a palindrome, taking into account strings of varying lengths (even and odd) and various types of characters (letters, digits, special characters).

Program:

```

def palindrome(s):
    if s.isalnum():
        if len(s)%2==0:
            if s==s[::-1]:
                return 1
            else:
                return "no"
        elif len(s)%2 !=0:
            if s==s[::-1]:
                return 0
            else:
                return "no"
    elif s.isalpha():
        if len(s)%2==0:
            if s==s[::-1]:
                return 1
            else:
                return "no"
        elif len(s)%2 !=0:
            if s==s[::-1]:
                return 0

```

```

else:
    return "no"

```

TestCase:

```

from q2 import palindrome

def test_alnum():
    assert palindrome("121") == 0
def test_alphanum():
    assert palindrome("racecar") == 0
def test_even_len():
    assert palindrome("123321") == 1
def test_odd_len():
    assert palindrome("refer") == 0

```

Expected Output:

```

===== test session starts =====
platform win32 -- Python 3.11.9, pytest-8.4.1, pluggy-1.6.0
rootdir: C:\Users\niran\Desktop\ML OPS\EX_!\q2
plugins: mock-3.14.1
collected 4 items

test_palindrome.py .... [100%]

===== 4 passed in 0.03s =====
PS C:\Users\niran\Desktop\ML OPS\EX_!\q2> █

```

3. Formulate and validate a test case for a Python function that determines the primality of a given number, encompassing both prime and non-prime numbers, as well as negative numbers and zero.

Program:

```

def primality(num):
    if num < 0:
        return "negative"
    elif num < 2:
        return 1
    elif num == 2:
        return "prime"
    elif num % 2 == 0:
        return "composite" # Explicit even number composite check
    i = 3
    while i * i <= num:
        if num % i == 0:
            return "composite"
        i += 2
    return "prime"

```

TestCase:

```

import unittest
from q3 import primality

class TestPrimalityFunction(unittest.TestCase):

    def test_negative_numbers(self):
        self.assertEqual(primality(-10), "negative")
        self.assertEqual(primality(-1), "negative")
        print("negative_numbers test case passed!!")

    def test_less_than_two(self):
        self.assertEqual(primality(0), 1)
        self.assertEqual(primality(1), 1)
        print("less_than_two passed!!")

    def test_prime_numbers(self):
        self.assertEqual(primality(2), "prime")
        self.assertEqual(primality(3), "prime")
        self.assertEqual(primality(13), "prime")
        self.assertEqual(primality(17), "prime")
        print("prime_numbers passed!!")

    def test_composite_numbers(self):
        self.assertEqual(primality(6), "composite")
        self.assertEqual(primality(9), "composite")
        self.assertEqual(primality(15), "composite")
        self.assertEqual(primality(25), "composite")
        print("composite_numbers passed!!")

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

```

Expected Output:

```

PS C:\Users\niran\Desktop\ML OPS\EX_!\q3> pytest test_pre.py
=====
test session starts =====
platform win32 -- Python 3.11.9, pytest-8.4.1, pluggy-1.6.0
rootdir: C:\Users\niran\Desktop\ML OPS\EX_!\q3
plugins: mock-3.14.1
collected 4 items

test_pre.py .... [100%]

===== 4 passed in 0.03s =====
PS C:\Users\niran\Desktop\ML OPS\EX_!\q3>

```

4.Create and assess a test case for a Python function that sorts a list of integers in ascending order, considering different list sizes, including empty lists and lists with repeated elements.

Program:

```
def sort_list (list):
    return sorted(list)
```

TestCase:

```
from q4 import sort_list
import unittest

class test_lists(unittest.TestCase):

    def test_even_size(self):
        l1=[2,7,19,20,45,12]
        self.assertEqual(sort_list(l1),[2,7,12,19,20,45])
    def test_odd_size(self):
        l2=[5,40,25,9,1]
        self.assertEqual(sort_list(l2),[1,5,9,25,40])
    def test_empty_list(self):
        l3=[]
        self.assertEqual(sort_list(l3),[])
    def test_negative_num(self):
        l4=[-7,-3,-20,-80,-23]
        self.assertEqual(sort_list(l4),[-80,-23,-20,-7,-3])
    def test_repeat_list(self):
        l5=[23,7,90,4,7,1,56,23]
        self.assertEqual(sort_list(l5),[1,4,7,7,23,23,56,90])

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

Expected Output:

```
PS C:\Users\niran\Desktop\ML OPS\EX_!> cd q4
PS C:\Users\niran\Desktop\ML OPS\EX_!\\q4> python list_sortlist.py
.....
-----
Ran 5 tests in 0.000s

OK
PS C:\Users\niran\Desktop\ML OPS\EX_!\\q4>
```

5.Develop a test case for a Python function that calculates the Fibonacci sequence up to a given number, ensuring comprehensive coverage of edge cases, including both positive and negative input values, as well as zero.

Program:

```
def fibonacci_upto(n):
    if n < 0:
        raise ValueError("Input must be a non-negative integer")
    if n == 0:
        return []
    if n == 1:
        return [0]
    seq = [0, 1]
    while seq[-1] + seq[-2] <= n:
        seq.append(seq[-1] + seq[-2])
    return seq
```

TestCase:

```
import unittest
from fibo import fibonacci_upto
class TestFibonacciFunction(unittest.TestCase):

    def test_positive_input(self):
        self.assertEqual(fibonacci_upto(1), [0])
        self.assertEqual(fibonacci_upto(5), [0, 1, 1, 2, 3, 5])
        self.assertEqual(fibonacci_upto(10), [0, 1, 1, 2, 3, 5, 8])

    def test_zero_input(self):
        self.assertEqual(fibonacci_upto(0), [])

    def test_one_input(self):
        self.assertEqual(fibonacci_upto(1), [0])

    def test_negative_input(self):
        with self.assertRaises(ValueError):
            fibonacci_upto(-5)

    def test_large_input(self):
        result = fibonacci_upto(100)
        expected = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
        self.assertEqual(result, expected)

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

Expected Output:

```
PS C:\Users\niran\Desktop\ML OPS\EX_!> cd q5
PS C:\Users\niran\Desktop\ML OPS\EX_!\\q5> python test_fibo.py
-----
Ran 5 tests in 0.001s

OK
PS C:\Users\niran\Desktop\ML OPS\EX_!\\q5>
```

Result :

All five Python functions — factorial, palindrome check, prime test, sorting, and Fibonacci — were implemented and tested using unittest and pytest with normal, boundary, negative, and special case inputs. After minor corrections, all test cases passed successfully, confirming correctness and robustness.