unittest

1. Write a unittest test case to verify the behavior of a class that implements basic bank account operations (deposit, withdraw, check balance).

```
class Bankaccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("deposit amount must be positive")
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("withdrawal amount must be positive")
        if amount > self.balance:
            raise ValueError("insufficient balance")
        self.balance -= amount
        return self.balance

    def get_balance(self):
        return self.balance
```

```
import unittest
from Bank import *
class TestBankAccount(unittest.TestCase):
    def setUp(self):
        self.acc = Bankaccount("raksha",1000)
    def test_initial_balance(self):
        self.assertEqual(self.acc.get_balance(),1000)
    def test_deposit(self):
        self.acc.deposit(500)
        self.assertEqual(self.acc.get_balance(),1500)
    def test_withdraw(self):
        self.acc.withdraw(300)
        self.assertEqual(self.acc.get_balance(),700)
    def test_withdraw_insufficient_funds(self):
        with self.assertRaises(ValueError):
            self.acc.withdraw(2000)
    def test_invalid_deposit(self):
        with self.assertRaises(ValueError):
            self.acc.deposit(-100)
    def test_invalid_withdraw(self):
        with self.assertRaises(ValueError):
            self.acc.withdraw(0)
if __name__ == '__main__':
    unittest.main()
```

2. How can you test private methods or variables in Python using unittest?

```
class Calculator:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __add(self):
        return self.a + self.b

    def __sub(self):
        return self.a - self.b

    def __mul(self):
        return self.a * self.b

    def __div(self):
        if self.b == 0:
            return "Cannot divide by zero"
        return self.a / self.b
```

```
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator(10, 5)

    def test_private_add(self):
        result = self.calc._Calculator__add()
        self.assertEqual(result, 15)

    def test_private_sub(self):
        result = self.calc._Calculator__sub()
        self.assertEqual(result, 5)

    def test_private_mul(self):
        result = self.calc._Calculator__mul()
        self.assertEqual(result, 50)

    def test_private_div(self):
        result = self.calc._Calculator__div()
        self.assertEqual(result, 2)

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

3. What is the role of setUpClass() and tearDownClass() in unittest? Provide a code snippet where these are useful.

- setUpClass(): Runs once before all tests in the class.
- tearDownClass(): Runs once after all tests in the class.

```
import unittest

class TestListOperations(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        print("\n🔧 setUpClass: Creating a sample list")
        cls.numbers = [10, 20, 30, 40, 50]

    @classmethod
    def tearDownClass(cls):
        print("🧹 tearDownClass: Clearing the list")
        cls.numbers = None

    def test_list_length(self):
        self.assertEqual(len(self.numbers), 5)

    def test_list_sum(self):
        self.assertEqual(sum(self.numbers), 150)

    def test_list_contains(self):
        self.assertIn(30, self.numbers)

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

4. How do you test a function that raises different exceptions based on input (e.g., ValueError for negatives, TypeError for strings)?

```
def check_number(x):
    if isinstance(x, str):
        raise TypeError("String is not allowed")
    if x < 0:
        raise ValueError("Negative number not allowed")
    return x * 2
```

```
import unittest
from validate import check_number

class TestCheckNumber(unittest.TestCase):

    def test_negative_number_raises_value_error(self):
        with self.assertRaises(ValueError):
            check_number(-5)

    def test_string_input_raises_type_error(self):
        with self.assertRaises(TypeError):
```

```
            check_number("abc")

    def test_valid_number(self):
        result = check_number(10)
        self.assertEqual(result, 20)

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

5. Write a unittest case to validate a function that returns the factorial of a number. Test valid inputs, 0, and invalid types.

```
def factorial(n):
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    if n == 0:
        return 1
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

```
import unittest
from math_utils import factorial

class TestFactorial(unittest.TestCase):

    def test_factorial_positive(self):
        self.assertEqual(factorial(5), 120)
        self.assertEqual(factorial(3), 6)

    def test_factorial_zero(self):
        self.assertEqual(factorial(0), 1)

    def test_factorial_negative(self):
        with self.assertRaises(ValueError):
            factorial(-4)

    def test_factorial_non_integer(self):
        with self.assertRaises(TypeError):
            factorial("abc")
        with self.assertRaises(TypeError):
            factorial(4.5)

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

6. How would you use unittest.skip, skipIf, or skipUnless in practical test cases?

```
import platform
import unittest
import sys


class TestmathFunctions(unittest.TestCase):
    @unittest.skip("test is skipped temporarily for debugging")
    def test_add(self):
        self.assertEqual(1 + 2, 3)

    @unittest.skipIf(sys.version_info < (3, 9), "requires python 3.9 or
higher")
    def test_dict_union_operator(self):
        a = {"x": 1}
        b = {"y": 2}
        self.assertEqual(a | b, {"x": 1, "y": 2})

    @unittest.skipUnless(platform.system() == "windows", "runs on windows")
    def test_windows_path(self):
        self.assertIn("c:", "c:\\ProgramFiles")


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

pytest

1. Write a pytest test using @pytest.mark.parametrize for a function that checks if a number is even.

```
import pytest

def is_even(n):
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")
    return n % 2 == 0

@pytest.mark.parametrize("num, expected", [
    (2, True),
    (3, False),
    (0, True),
    (-4, True),
    (-5, False)
])
def test_is_even_valid(num, expected):
    assert is_even(num) == expected

def test_is_even_type_error():
    with pytest.raises(TypeError):
        is_even("ten")
```

2. How can you use a fixture in pytest to provide test data to multiple test functions?

```python
import pytest

class Calculator:
    def add(self, a, b): return a + b
    def sub(self, a, b): return a - b
    def mul(self, a, b): return a * b
    def div(self, a, b):
        if b == 0:
            raise ZeroDivisionError("Cannot divide by zero")
        return a / b

@pytest.fixture
def calc_data():
    calc = Calculator()
    a, b = 10, 5
    return calc, a, b

def test_add(calc_data):
    calc, a, b = calc_data
    assert calc.add(a, b) == 15

def test_sub(calc_data):
    calc, a, b = calc_data
    assert calc.sub(a, b) == 5

def test_mul(calc_data):
    calc, a, b = calc_data
    assert calc.mul(a, b) == 50

def test_div(calc_data):
    calc, a, b = calc_data
    assert calc.div(a, b) == 2
```

3. Use pytest.raises to test a function that throws a ValueError when a string input is passed to a numeric-only function.

```python
import pytest

# Division function to test
def divide(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise ValueError("Inputs must be numbers")
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    return a / b

# Test: valid input
def test_divide_valid():
    assert divide(10, 2) == 5
    assert divide(7.5, 2.5) == 3.0
```

```
# Test: non-numeric input
def test_divide_value_error():
    with pytest.raises(ValueError, match="Inputs must be numbers"):
        divide("a", 5)

# Test: division by zero
def test_divide_zero_division():
    with pytest.raises(ZeroDivisionError, match="Cannot divide by zero"):
        divide(10, 0)
```

4. Demonstrate how pytest.mark.skipif can be used to conditionally skip a test if the Python version is < 3.9.

```
import platform
import pytest
import sys
def test_addition():
     assert 1 + 2 == 3

@pytest.mark.skipif(sys.version_info < (3, 9), reason="requires python 3.9 or
higher")
def test_dict_merge_operator():
    a = {"x": 1}
    b = {"y": 2}
    result = a | b
    assert result == {"x": 1, "y": 2}
```

5. Create a test using pytest where the test fails and is marked as expected to fail using @pytest.mark.xfail.

```
import pytest

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

# This test will FAIL (division by zero), but it's marked as expected to fail
@pytest.mark.xfail(reason="Division by zero is not allowed")
def test_divide_by_zero():
    assert divide(10, 0) == 0  # This will raise ZeroDivisionError

# This test will pass
def test_divide_valid():
    assert divide(10, 2) == 5
```

6. How do you use tmp_path or tmpdir fixture in pytest to test functions that create or write to files?

```python
def write_hello(file_path):
    with open(file_path, 'w') as f:
        f.write("Hello, World!")

def read_content(file_path):
    with open(file_path, 'r') as f:
        return f.read()

def test_file_write_and_read(tmp_path):
    # Create a file path inside the temporary directory
    file = tmp_path / "greeting.txt"

    # Call the function to write to file
    write_hello(file)

    # Read and verify content
    content = read_content(file)
    assert content == "Hello, World!"
```