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 TestBank(unittest.TestCase):

def setUp(self):

self.acc= Bankaccount("abhigna",1000)

def test\_initial\_balance(self):

self.assertEqual(self.acc.get\_balance(), 1000)

def test\_deposit(self):

self.acc.deposit(1000)

self.assertEqual(self.acc.get\_balance(), 2000)

def test\_withdraw(self):

self.acc.withdraw(400)

self.assertEqual(self.acc.get\_balance(), 600)

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 Student:

def \_\_init\_\_(self, name, marks):

self.name = name

self.marks = marks

def \_\_result(self): # private method

if self.marks >= 35:

return f"{self.name} passed"

else:

return f"{self.name} failed"

def get\_resul(self): # public method

return self.\_\_result()

import unittest

from student import Student

class TestStudent(unittest.TestCase):

def test\_get\_result\_pass(self):

s = Student("Abhi", 95)

result = s.\_Student\_\_result()

self.assertEqual(result, "Abhi passed")

def test\_get\_result\_fail(self):

s = Student("Bhumi", 20)

result = s.\_Student\_\_result()

self.assertEqual(result, "Bhumi failed")

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()-> this runs once before any test methods defined in the class.

tearDownClass()-> it runs after all tests/test cases(methods)defined in the class.

class TestExample(unittest.TestCase):

@classmethod

def setUpClass(cls):

print("Setup all test cases")

@classmethod

def tearDownClass(cls):

print("Teardown all test cases")

def test1(self):

self.assertFalse(False)

def test2(self):

self.assertEqual(2 \* 2, 4)

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

def input(x):

if isinstance(x, str):

raise TypeError("string value is not a valid input")

if x < 0:

raise ValueError("negative numbers is not a valid input")

return x \* 5

if x == 0:

raise ZeroDivisionError("Cannot divide by zero")

return x / 5

from class import input

class TestInput(unittest.TestCase):

def test\_string(self):

with self.assertRaises(TypeError):

input("abhi")

def test\_negative(self):

with self.assertRaises(ValueError):

input(-5)

def test\_zero\_division(self):

with self.assertRaises(ZeroDivisionError):

input(10, 0)

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("integers is valid input")

if n < 0:

raise ValueError("negative numbers is not a valid input")

return 1 if n == 0 else n \* factorial(n - 1)

from class import factorial

class TestFactorial(unittest.TestCase):

def test\_valid(self):

self.assertEqual(factorial(4), 24)

def test\_zero(self):

self.assertEqual(factorial(0), 1)

def test\_type\_error(self):

with self.assertRaises(TypeError):

factorial('a')

def test\_negative(self):

with self.assertRaises(ValueError):

factorial(-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()

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):

return n % 2 == 0

@pytest.mark.parametrize('num,result',[(2,True),(3,False),(4,True),(5,False),(6,True),(7,False),(8,True),(9,False)])

def test\_is\_even(num,result):

assert result == is\_even(num)

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

@pytest.fixture

def sample\_data():

return {"name " :"abhi" , "technology" :"python"}

def test\_sample\_data(sample\_data):

assert sample\_data["name "] == "abhi"

#exceptions

def divide(a,b):

return a/b

def test\_divide():

with pytest.raises(ZeroDivisionError):

divide(10,0)

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

def square(a):

if isinstance(a, str):

raise ValueError("input which is string is not valid/allowed")

return a \* a

def test\_square\_raises():

with pytest.raises(ValueError):

square("abhi")

def test\_number():

assert square(5)==5

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

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.

@pytest.mark.xfail

def test\_fail():

def add(a, b):

return a + b

def test\_add():

assert add(2, 3) == 7

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

def write\_file(path, content):

with open(path, 'w') as file:

file.write(content)

def test\_write\_file(tmp\_path):

file = tmp\_path / "test.txt"

write\_file(file, "abhigna")

assert file.read\_text() == "abhigna"