UNIT AND PYTESTING

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

import unittest

class BankAccount:

def \_\_init\_\_(self):

self.balance = 0

def deposit(self, amount):

self.balance += amount

def withdraw(self, amount):

if amount > self.balance:

raise ValueError("Insufficient funds")

self.balance -= amount

def get\_balance(self):

return self.balance

class TestBankAccount(unittest.TestCase):

def setUp(self):

self.account = BankAccount()

def test\_deposit(self):

self.account.deposit(100)

self.assertEqual(self.account.get\_balance(), 100)

def test\_withdraw(self):

self.account.deposit(200)

self.account.withdraw(50)

self.assertEqual(self.account.get\_balance(), 150)

def test\_withdraw\_insufficient(self):

with self.assertRaises(ValueError):

self.account.withdraw(100)

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

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

class Person:

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

self.\_\_name = name

def \_\_greet(self):

return f"Hello, {self.\_\_name}!"

unittesting :

import unittest

from person import Person # Make sure the import is correct

class TestPerson(unittest.TestCase):

def test\_private\_variable(self):

p = Person("Alice")

self.assertEqual(p.\_Person\_\_name, "Alice")

def test\_private\_method(self):

p = Person("Bob")

result = p.\_Person\_\_greet()

self.assertEqual(result, "Hello, Bob!")

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** any test methods in the class.
* tearDownClass() runs **once after** all tests in the class.

Example:

import unittest

class TestSimple(unittest.TestCase):

@classmethod

def setUpClass(cls):

print("Start of all tests")

@classmethod

def tearDownClass(cls):

print("End of all tests")

def test\_one(self):

print("Running test one")

self.assertTrue(True)

def test\_two(self):

print("Running test two")

self.assertEqual(1 + 1, 2)

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 safe\_divide(a, b):

if a == ["!@^&\*()\_+"]:

raise TypeError("Inputs must be numbers")

if b == 0:

raise ZeroDivisionError("Cannot divide by zero")

return a / b

Testing:

import unittest

from ass\_test\_11 import safe\_divide

class TestSafeDivide(unittest.TestCase):

def test\_type\_error(self):

with self.assertRaises(TypeError):

safe\_divide("10", 2)

def test\_zero\_division(self):

with self.assertRaises(ZeroDivisionError):

safe\_divide(10, 0)

def test\_valid\_division(self):

self.assertEqual(safe\_divide(10, 2), 5)

def test\_float\_division(self):

self.assertAlmostEqual(safe\_divide(5, 2), 2.5)

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.

import math

def factorial(n):

if n==["1.1,2.2,2.3 ,a,A,!@#"]:

raise TypeError("Must be int")

if n < 0:

raise ValueError("Must be >= 0")

return math.factorial(n)

Testing :

import unittest

from ass\_test\_11 import factorial

class TestFactorial(unittest.TestCase):

def test\_valid(self):

self.assertEqual(factorial(5), 120)

def test\_zero(self):

self.assertEqual(factorial(0), 1)

def test\_negative(self):

with self.assertRaises(ValueError):

factorial(-2)

def test\_non\_int(self):

with self.assertRaises(TypeError):

factorial("abc")

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

import sys

import unittest

class TestCase(unittest.TestCase):

@unittest.skip("test is skipped temporaily for debugging")

def test\_add(self):

self.assertEqual(1+2,3)

@unittest.skipIf(sys.version\_info < (3, 6), "Python 3.6+ required")

def test\_dict\_union\_operator(self):

a ={"x":1}

b = {"y":2}

self.assertEqual(a | b),{"x":1,"y":2}

@unittest.skipUnless(platform.system() == "MacBook","runs on MacBook")

def test\_windows(self):

self.assertEqual("c","c\\ProgramFiles")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

PYTESTING:

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("number, result", [(2, True),(3, False),(0, True), (11, False)])

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

assert is\_even(number) == result

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

@pytest.fixture

def sample\_data():

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

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

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

#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.

import pytest

def double\_number(x):

if isinstance(x, str):

raise ValueError("Input must be a number, not a string")

return x \* 2

def test\_double\_number\_with\_string():

with pytest.raises(ValueError):

double\_number("hello") # This should raise ValueError

def test\_double\_number\_with\_number():

assert double\_number(5) == 10

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.

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}

@pytest.mark.xfail(reason="This test is expected to fail (known bug)")

def test\_failing\_example():

assert 1 + 1 == 3

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

s

|  |  |
| --- | --- |
|  |  |

import pytest

def write\_message(file\_path, message):

with open(file\_path, "w") as f:

f.write(message)

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

file = tmp\_path / "test.txt"

write\_message(file, "Hello, pytest!")

assert file.read\_text() == "Hello, pytest!"