**Day-11 Evening Assessment**

**Unittest**

1. import unittest  
  
class BankAccount:  
   def \_\_init\_\_(self, balance=0):  
       self.\_balance = balance  
  
   def deposit(self, amount):  
       if amount < 0:  
           raise ValueError("Deposit amount must be positive")  
       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(100)  
  
   def test\_deposit(self):  
       self.account.deposit(50)  
       self.assertEqual(self.account.get\_balance(), 150)  
  
   def test\_withdraw(self):  
       self.account.withdraw(30)  
       self.assertEqual(self.account.get\_balance(), 70)  
  
   def test\_withdraw\_insufficient\_funds(self):  
       with self.assertRaises(ValueError):  
           self.account.withdraw(200)  
  
   def test\_deposit\_negative(self):  
       with self.assertRaises(ValueError):  
           self.account.deposit(-50)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
   unittest.main()  
  
2. Private methods/variables should ideally not be accessed directly in tests, you can access them:  
Single underscore \_var: Use directly.  
Double underscore \_\_var: Use name mangling: \_ClassName\_\_var

3. They are class-level setup/teardown methods. Useful for resource-intensive setup (like DB connections or test files) that should run once per test class.

import unittest  
class TestWithSetupClass(unittest.TestCase):  
 @classmethod  
 def setUpClass(cls):  
 print("Set up shared resources once before all tests")  
  
 @classmethod  
 def tearDownClass(cls):  
 print("Clean up resources after all tests run")  
  
 def test\_one(self):  
 self.assertTrue(True)  
  
 def test\_two(self):  
 self.assertEqual(1 + 1, 2)

4. import unittest  
def check\_input(x):  
 if isinstance(x, str):  
 raise TypeError("String not allowed")  
 if x < 0:  
 raise ValueError("Negative not allowed")  
 return x \* 2  
  
class TestCheckInput(unittest.TestCase):  
 def test\_valid\_input(self):  
 self.assertEqual(check\_input(5), 10)  
  
 def test\_negative\_input(self):  
 with self.assertRaises(ValueError):  
 check\_input(-1)  
  
 def test\_string\_input(self):  
 with self.assertRaises(TypeError):  
 check\_input("abc")

5. import unittest  
def factorial(n):  
 if not isinstance(n, int):  
 raise TypeError("Input must be integer")  
 if n < 0:  
 raise ValueError("Negative not allowed")  
 return 1 if n == 0 else n \* factorial(n - 1)  
  
class TestFactorial(unittest.TestCase):  
 def test\_valid\_inputs(self):  
 self.assertEqual(factorial(5), 120)  
 self.assertEqual(factorial(3), 6)  
  
 def test\_zero(self):  
 self.assertEqual(factorial(0), 1)  
  
 def test\_negative\_input(self):  
 with self.assertRaises(ValueError):  
 factorial(-2)  
  
 def test\_invalid\_type(self):  
 with self.assertRaises(TypeError):  
 factorial("abc")

6. They allow conditional skipping of tests. Useful for:  
-Feature flags  
-OS/platform-specific behavior  
-Incomplete or temporarily broken tests  
  
import sys  
import unittest  
  
class TestSkips(unittest.TestCase):  
  
 @unittest.skip("Test skipped unconditionally")  
 def test\_skip\_always(self):  
 self.assertTrue(False)  
  
 @unittest.skipIf(sys.platform.startswith("win"), "Skip on Windows")  
 def test\_not\_on\_windows(self):  
 self.assertTrue(True)  
  
 @unittest.skipUnless(sys.platform.startswith("linux"), "Run only on Linux")  
 def test\_only\_on\_linux(self):  
 self.assertTrue(True)

**Pytest**

1. import pytest  
  
def is\_even(n):  
   return n % 2 == 0  
  
@pytest.mark.parametrize("input\_val,expected", [  
   (2, True),  
   (3, False),  
   (0, True),  
   (-4, True),  
   (-1, False)  
])  
def test\_is\_even(input\_val, expected):  
   assert is\_even(input\_val) == expected  
  
2. The sample\_data fixture is shared across both tests automatically.  
import pytest  
  
  
@pytest.fixture  
def sample\_data():  
 return [1, 2, 3, 4, 5]  
  
def test\_sum(sample\_data):  
 assert sum(sample\_data) == 15  
  
def test\_max(sample\_data):  
 assert max(sample\_data) == 5

3. import pytest  
  
def square(n):  
   if not isinstance(n, (int, float)):  
       raise ValueError("Only numbers are allowed")  
   return n \* n  
  
def test\_square\_with\_invalid\_input():  
   with pytest.raises(ValueError, match="Only numbers are allowed"):  
       square("string")  
  
4. import sys  
import pytest  
  
@pytest.mark.skipif(sys.version\_info < (3, 9), reason="Requires Python 3.9 or higher")  
def test\_feature\_requiring\_python39():  
   assert (1 << 256).bit\_length() > 0  
  
5. import pytest  
  
def divide(a, b):  
   return a / b  
  
@pytest.mark.xfail(reason="Division by zero should fail")  
def test\_divide\_by\_zero():  
   divide(1, 0)  
This test will fail, but pytest marks it as expected failure.  
  
6. def write\_hello(file\_path):  
   with open(file\_path, "w") as f:  
       f.write("Hello, pytest!")  
  
def test\_write\_hello(tmp\_path):  
   file = tmp\_path / "greeting.txt"  
   write\_hello(file)  
  
   assert file.read\_text() == "Hello, pytest!"