# DAY-11 evening assessment.

# unittest

1.

account  
class BankAccount:  
 def \_\_init\_\_(self,initial\_balance=0):  
 self.balance = initial\_balance  
  
 def deposit(self,amount):  
 if amount <= 0:  
 raise ValueError("Deposit amount must be greater than zero")  
 self.balance += amount  
  
 def withdraw(self,amount):  
 if amount <= 0:  
 raise ValueError("Withdraw amount must be greater than zero")  
 if amount>self.balance:  
 raise ValueError("Withdraw amount cannot be greater than balance")  
 self.balance -= amount  
  
 def check\_balance(self):  
 return self.balance

#unit test file of bank\_account  
import unittest  
from bank\_account import BankAccount  
  
class TestBankAccount(unittest.TestCase):  
 def setUp(self):  
 self.account=BankAccount(100)  
  
 def test\_initial\_balance(self):  
 self.assertEqual(self.account.check\_balance(), 100)  
  
 def test\_deposit(self):  
 self.account.deposit(50)  
 self.assertEqual(self.account.check\_balance(), 150)  
  
 def test\_deposit\_invalid\_amount(self):  
 with self.assertRaises(ValueError):  
 self.account.deposit(-20)  
  
 def test\_withdraw(self):  
 self.account.withdraw(30)  
 self.assertEqual(self.account.check\_balance(), 70)  
  
 def test\_withdraw\_insufficient\_funds(self):  
 with self.assertRaises(ValueError):  
 self.account.withdraw(200)  
  
 def test\_withdraw\_invalid\_amount(self):  
 with self.assertRaises(ValueError):  
 self.account.withdraw(0)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

2.by using name mangling we can test private methods and variables in python using unittest.

3.They are used in cases like database connection ,file reading etc.

These are class level setup and teardown methods in unit test. They run once before and after all test methods in the class.

import unittest  
class DatabaseTest(unittest.TestCase):  
 @classmethod  
 def setUpClass(cls):  
 print("connecting to database")  
 cls.db={"users":["Alice","Bob"]}  
   
 @classmethod  
 def tearDownClass(cls):  
 print("disconnecting from database")  
   
 def test\_user\_exists(self):  
 self.assertIn("Alice",self.db["users"])  
   
 def test\_user\_does\_not\_exist(self):  
 self.assertNotIn("Carlie",self.db["users"])

4.using assertRaises in unittest.

def process\_number(x):  
 if isinstance(x, str):  
 raise TypeError('string is not allowed')  
 if x<0:  
 raise ValueError('negative numbers are not allowed')  
 return x\*2  
  
import unittest  
  
class TestExceptions(unittest.TestCase):  
 def test\_value\_error(self):  
 with self.assertRaises(ValueError):  
 process\_number(-1)  
  
 def test\_type\_error(self):  
 with self.assertRaises(TypeError):  
 process\_number("abc")

5. def factorial(n):  
   if not isinstance(n, int):  
       raise TypeError("Only integers allowed")  
   if n < 0:  
       raise ValueError("Negative not allowed")  
   if n == 0:  
       return 1  
   result = 1  
   for i in range(1, n + 1):  
       result \*= i  
   return result  
  
import unittest  
  
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(-3)  
  
   def test\_invalid\_type(self):  
       with self.assertRaises(TypeError):  
           factorial("five")

6. import unittest  
import sys  
  
class TestSkipExamples(unittest.TestCase):  
   @unittest.skip("Skip this unconditionally")  
   def test\_skip(self):  
       self.fail("Should be skipped")  
  
   @unittest.skipIf(sys.platform.startswith("win"), "Skip on Windows")  
   def test\_skip\_if(self):  
       self.assertTrue(True)  
  
   @unittest.skipUnless(sys.version\_info >= (3, 9), "Needs Python >= 3.9")  
   def test\_skip\_unless(self):  
       self.assertTrue(True)

# Pytest

1. import pytest  
  
def is\_even(n):  
   return n % 2 == 0  
  
@pytest.mark.parametrize("num,expected", [  
   (2, True),  
   (3, False),  
   (0, True),  
   (-4, True)  
])  
def test\_is\_even(num, expected):  
   assert is\_even(num) == expected

2. import pytest  
  
@pytest.fixture  
def sample\_list():  
   return [1, 2, 3, 4]  
  
def test\_length(sample\_list):  
   assert len(sample\_list) == 4  
  
def test\_sum(sample\_list):  
   assert sum(sample\_list) == 10

3. import pytest  
  
def square\_number(x):  
   if isinstance(x, str):  
       raise ValueError("String not allowed")  
   return x \*\* 2  
  
def test\_square\_number\_raises():  
   with pytest.raises(ValueError):  
       square\_number("ten")

4. import pytest  
import sys  
  
@pytest.mark.skipif(sys.version\_info < (3, 9), reason="Requires Python 3.9+")  
def test\_feature\_39():  
   # e.g., using dict union operator (Python 3.9+)  
   d1 = {"a": 1}  
   d2 = {"b": 2}  
   assert d1 | d2 == {"a": 1, "b": 2}

5. import pytest  
  
@pytest.mark.xfail(reason="Bug not fixed yet")  
def test\_fail\_case():  
   assert 1 + 1 == 3

6. def write\_to\_file(file\_path, content):  
   with open(file\_path, "w") as f:  
       f.write(content)  
  
def test\_write\_to\_file(tmp\_path):  
   # Create a path to a temporary file  
   test\_file = tmp\_path / "output.txt"  
  
   # Call the function that writes to the file  
   write\_to\_file(test\_file, "Hello, World!")  
  
   # Assert the file exists and content is correct  
   assert test\_file.exists()  
   assert test\_file.read\_text() == "Hello, World!"