AI ASSISTED CODING

LAB TEST – 02

NAME : Neha Shazneen

ROLL NO : 2403A510A1

BATCH : 05

DEPARTMENT : CSE

SET – 03

Question 1 : AI-Assisted Unit Test Generation

TASK 1 : Provide a Python function (e.g., calculate_area(radius)) to the AI and ask it to generate unit tests using unittest or pytest.

Prompt :

```
def calculate_area(radius):
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    import math
    return math.pi * radius * radius
```

Please generate unit tests for this function using unittest (or pytest). Include tests for:

A valid radius

Zero radius

Negative radius (should raise ValueError)

Provide the complete test code.

Code Generated :

```python
import math
import unittest

def calculate_area(radius):
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius * radius

class TestCalculateArea(unittest.TestCase):

    def test_positive_radius(self):
        result = calculate_area(5)
        expected = math.pi * 25
        self.assertAlmostEqual(result, expected)

    def test_zero_radius(self):
        result = calculate_area(0)
        expected = 0
        self.assertEqual(result, expected)

    def test_negative_radius(self):
        with self.assertRaises(ValueError):
            calculate_area(-3)

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

Output :

```
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>  & 'c:\User
s\Summya Mehveen\AppData\Local\Programs\Python\Python313\python.exe' 'c:\User
s\Summya Mehveen\.cursor\extensions\ms-python.debugpy-2025.10.0-win32-x64\bun
dled\libs\debugpy\launcher' '49689' '--' 'c:\Users\Summya Mehveen\OneDrive\De
sktop\AI associated coding\lab2q1t1.py'
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>
```

Observation :

The test cases verify that the calculate_area function behaves correctly for different types of input values.

1. For a **positive radius**, the function correctly calculates the area using the formula πr^2.
2. For a **radius of zero**, the function returns 0, which is the expected mathematical result.
3. For a **negative radius**, the function correctly raises a ValueError, ensuring invalid input is handled safely.

All test cases passed successfully, confirming that the function is robust and handles both valid and invalid inputs as expected.

TASK 2 : Run the generated tests, analyze test coverage, and modify the AI prompt to include edge cases (e.g., negative radius).

Prompt :

Run the existing unit tests for the function calculate_area and analyze the results. Check test coverage and identify any missing cases. Update the test suite to include more edge cases like negative radius, zero, very large values, floating-point values, and very small positive values. Provide the improved test code using unittest.

Code Generated :

```python
import math
import unittest
import sys

def calculate_area(radius):
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius * radius


class TestCalculateArea(unittest.TestCase):

    def test_positive_integer_radius(self):
        result = calculate_area(5)
        expected = math.pi * 25
        self.assertAlmostEqual(result, expected, places=10)

    def test_positive_float_radius(self):
        result = calculate_area(2.5)
        expected = math.pi * 2.5 * 2.5
        self.assertAlmostEqual(result, expected, places=10)

    def test_zero_radius(self):
        self.assertEqual(calculate_area(0), 0)

    def test_negative_radius(self):
        with self.assertRaises(ValueError) as context:
            calculate_area(-1)
        self.assertEqual(str(context.exception), "Radius cannot be negative")
```

```python
    def test_very_small_positive_radius(self):
        small_value = 1e-10
        result = calculate_area(small_value)
        expected = math.pi * small_value * small_value
        self.assertAlmostEqual(result, expected, places=15)

    def test_very_large_radius(self):
        large_value = 10**6
        result = calculate_area(large_value)
        expected = math.pi * large_value * large_value
        self.assertAlmostEqual(result, expected, places=5)

    def test_extremely_large_radius(self):
        large_value = 10**15
        result = calculate_area(large_value)
        expected = math.pi * large_value * large_value
        self.assertAlmostEqual(result, expected, places=0)

    def test_precision_with_decimal_places(self):
        test_cases = [1.1, 3.14159, 2.71828, 0.123456789]
        for radius in test_cases:
            with self.subTest(radius=radius):
                result = calculate_area(radius)
                expected = math.pi * radius * radius
                self.assertAlmostEqual(result, expected, places=10)

```

```
    def test_boundary_around_zero(self):
        small_values = [1e-15, 1e-12, 1e-9, 1e-6, 1e-3]
        for value in small_values:
            with self.subTest(value=value):
                result = calculate_area(value)
                expected = math.pi * value * value
                self.assertAlmostEqual(result, expected, places=15)

    def test_negative_radius_edge_cases(self):
        negative_values = [-0.1, -1, -10, -100, -1e-6]
        for value in negative_values:
            with self.subTest(value=value):
                with self.assertRaises(ValueError):
                    calculate_area(value)

    def test_radius_one(self):
        result = calculate_area(1)
        self.assertAlmostEqual(result, math.pi, places=10)

    def test_radius_sqrt_pi(self):
        radius = math.sqrt(math.pi)
        result = calculate_area(radius)
        expected = math.pi * math.pi
        self.assertAlmostEqual(result, expected, places=10)

```

```
82      def test_consistency_with_math_pi(self):
83          radius = 2.0
84          result = calculate_area(radius)
85          manual_calculation = math.pi * radius * radius
86          self.assertEqual(result, manual_calculation)
87
88      def test_return_type(self):
89          result = calculate_area(5)
90          self.assertIsInstance(result, float)
91
92      def test_performance_large_calculation(self):
93          import time
94          start_time = time.time()
95          result = calculate_area(10**8)
96          end_time = time.time()
97          self.assertLess(end_time - start_time, 1.0)
98          expected = math.pi * (10**8) * (10**8)
99          self.assertAlmostEqual(result, expected, places=0)
100
101
102 def run_tests_with_coverage():
103     try:
104         import coverage
105         cov = coverage.Coverage()
106         cov.start()
107
108         unittest.main(exit=False, verbosity=2)
```

```
109
110             cov.stop()
111             cov.save()
112
113             print("\n" + "="*50)
114             print("COVERAGE REPORT")
115             print("="*50)
116             cov.report()
117
118         except ImportError:
119             print("Coverage module not available. Install with: pip install coverage")
120             unittest.main(verbosity=2)
121
122     if __name__ == "__main__":
123         print("Running comprehensive tests for calculate_area function...")
124         print("="*60)
125
126         unittest.main(verbosity=2, exit=False)
127
128         print("\n" + "="*60)
129         print("Test Summary:")
130         print("- All basic functionality tests")
131         print("- Edge cases and boundary conditions")
132         print("- Error handling tests")
133         print("- Precision and performance tests")
134         print("- Large value handling tests")
135         print("="*60)
```

Output :

```
Ran 15 tests in 0.006s

OK

============================================================
Test Summary:
- All basic functionality tests
- Edge cases and boundary conditions
- Error handling tests
- Precision and performance tests
- Large value handling tests
```

Observation :

The updated unit tests provide broader validation of the calculate_area function. The test suite now checks correctness across a wide range of input conditions including valid values, zero, negative values, floating-point inputs, very large numbers, and extremely small radius values. All valid inputs produce mathematically accurate results, while negative input correctly raises a

ValueError. The improved coverage ensures the function is reliable, handles edge cases properly, and behaves as expected under different scenarios.

QUESTION 2 : Test-Driven Enhancement

TASK 1 : Give the AI a partially implemented function (e.g., validate_password(password)) and ask it to create test cases first (TDD approach).

Prompt :

“You are given a partially implemented function validate_password(password).

Using the Test-Driven Development (TDD) approach, first create comprehensive test cases for this function using Python’s unittest framework.

Then, after tests are written, implement the function so that all test cases pass.

Finally, provide your observations.”

Code Generated :

```python
import unittest
import re

# --- Function to validate password ---
def validate_password(password):
    """
    Validates a password based on the following rules:
    - Minimum 8 characters
    - At least one uppercase letter
    - At least one lowercase letter
    - At least one digit
    - At least one special character (@, $, !, %, *, ?, &)
    """
    if len(password) < 8:
        return False
    if not re.search(r"[A-Z]", password):
        return False
    if not re.search(r"[a-z]", password):
        return False
    if not re.search(r"[0-9]", password):
        return False
    if not re.search(r"[@$!%*?&]", password):
        return False
    return True


```

```python
# --- Test cases for validate_password() ---
class TestValidatePassword(unittest.TestCase):
    def test_valid_password(self):
        self.assertTrue(validate_password("Valid@123"))

    def test_too_short(self):
        self.assertFalse(validate_password("Va@1"))

    def test_no_uppercase(self):
        self.assertFalse(validate_password("valid@123"))

    def test_no_lowercase(self):
        self.assertFalse(validate_password("VALID@123"))

    def test_no_digit(self):
        self.assertFalse(validate_password("Valid@Pass"))

    def test_no_special_char(self):
        self.assertFalse(validate_password("Valid1234"))

    def test_only_special_characters(self):
        self.assertFalse(validate_password("@@@@@@@@@@"))

    def test_empty_password(self):
        self.assertFalse(validate_password(""))

```

```python
# --- Main execution for unittest ---
if __name__ == "__main__":
    unittest.main()
```

Output :

```
........
----------------------------------------------------------------------
Ran 8 tests in 0.008s

OK
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>
```

Observation :

All the test cases passed successfully.

The validate_password() function works correctly and follows all the given password rules — length, uppercase, lowercase, digit, and special character.

Hence, the program runs successfully and meets the expected requirements.

TASK 2 : Use the generated test cases to drive code completion. Evaluate whether the final function passes all tests and meets design expectations.

Prompt :

“Use the previously generated unit test cases for the partially implemented function validate_password(password) to complete the code.

Implement the function so that it passes all the test cases.

After implementation, evaluate whether the final function satisfies all tests and meets the password validation design expectations.”

Code Generated :

```python
import unittest
import re

def validate_password(password):
    """
    Validates a password based on the following design expectations:
      ✅ Minimum 8 characters
      ✅ At least one uppercase letter
      ✅ At least one lowercase letter
      ✅ At least one digit
      ✅ At least one special character (@, $, !, %, *, ?, &)
    """
    if len(password) < 8:
        return False
    if not re.search(r"[A-Z]", password):   # Check for uppercase
        return False
    if not re.search(r"[a-z]", password):   # Check for lowercase
        return False
    if not re.search(r"[0-9]", password):   # Check for digit
        return False
    if not re.search(r"[@$!%*?&]", password):  # Check for special character
        return False
    return True

class TestValidatePassword(unittest.TestCase):
    def test_valid_password(self):
        self.assertTrue(validate_password("Valid@123"))

```

```
    def test_too_short(self):
        self.assertFalse(validate_password("Va@1"))

    def test_no_uppercase(self):
        self.assertFalse(validate_password("valid@123"))

    def test_no_lowercase(self):
        self.assertFalse(validate_password("VALID@123"))

    def test_no_digit(self):
        self.assertFalse(validate_password("Valid@Pass"))

    def test_no_special_char(self):
        self.assertFalse(validate_password("Valid1234"))

    def test_only_special_characters(self):
        self.assertFalse(validate_password("@@@@@@@@@@"))

    def test_empty_password(self):
        self.assertFalse(validate_password(""))


if __name__ == "__main__":
    print("Running test cases for validate_password()...\n")
    result = unittest.main(exit=False)

    if result.result.wasSuccessful():
        print("\n✅ All tests passed successfully!")
        print("✅ The function meets all password validation design expectations.")
    else:
        print("\n❌ Some tests failed. Please review the implementation.")
```

Output :

```
Running test cases for validate_password()...

........
----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK

✅ All tests passed successfully!
✅ The function meets all password validation design expectations.
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>
```

Observation :

All the test cases passed successfully after implementing the validate_password() function.

The function meets all the design rules for password validation.

This shows that the Test-Driven Development (TDD) approach helped create a correct and reliable solution.