K.Akshay
2303A51817
B-26

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:** 2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s)Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju Mr. G. Kranthi		
Course Code	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/I	**Regulation**	R23
Date and Day of Assignment	Week 4 - Thursday	**Time(s)**	23CSBTB01 To 23CSBTB52
Duration	2 Hours	**Applicable to Batches**	All Batches
AssignmentNumber:8.4 (Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*ExpectedTime to complete*
1	Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases	Week 4

Lab Objectives:

- To introduce students to test-driven development (TDD) using AI code generation tools.
- To enable the generation of test cases before writing code implementations.
- To reinforce the importance of testing, validation, and error handling.
- To encourage writing clean and reliable code based on AI-generated test expectations.

Lab Outcomes (LOs):

By the end of this lab, students will be able to:

- Apply TDD methodology using AI tools.
- Generate test cases before writing the actual code logic.
- Validate and refactor code based on test outcomes.
- Use Python's unittest or pytest libraries for test-driven development.
- Develop confidence in debugging and improving code with AI guidance.

Task 1: Developing a Utility Function Using TDD

Scenario
You are working on a small utility library for a larger software system. One of the required functions should calculate the square of a given number, and correctness is critical because other modules depend on it.

Task Description
Following the **Test Driven Development (TDD)** approach:

1. First, write unit test cases to verify that a function correctly returns the square of a number for multiple inputs.
2. After defining the test cases, use **GitHub Copilot or Cursor AI** to generate the function implementation so that all tests pass.

Ensure that the function is written **only after** the tests are created.

Expected Outcome

- A separate test file and implementation file
- Clearly written test cases executed before implementation
- AI-assisted function implementation that passes all tests
- Demonstration of the TDD cycle: *test → fail → implement → pass*

Prompt:-
First, create a file test_square.py and write unit tests for a function square(num) using Python unittest.
Test positive, negative, zero, and decimal numbers.
Do not write the function yet.
After the tests are written, create square.py and implement square(num) so that all tests pass.
Code:-

lab 8.4 > test_square.py > TestSquare > test_negative_number

```python
import unittest
from square import square

class TestSquare(unittest.TestCase):

    def test_positive_number(self):
        self.assertEqual(square(5), 25)
        self.assertEqual(square(3), 9)

    def test_negative_number(self):
        self.assertEqual(square(-5), 25)
        self.assertEqual(square(-3), 9)

    def test_zero(self):
        self.assertEqual(square(0), 0)

    def test_decimal_number(self):
        self.assertEqual(square(2.5), 6.25)
        self.assertEqual(square(1.5), 2.25)


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

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

----------------------------------------------------------------------
Ran 4 tests in 0.001s


OK
```

Explanation:-

The four dots indicate that four test cases were executed and passed.
Ran 4 tests means the program tested the function four times with different inputs.ok means there are no errors and the square() function is working correctly.

Task 2: Email Validation for a User Registration System

Scenario
You are developing the backend of a user registration system. One requirement is to validate user email addresses before storing them in the database.

Task Description
Apply **Test Driven Development** by:

1. Writing unit test cases that define valid and invalid email formats (e.g., missing @, missing domain, incorrect structure).
2. Using **AI assistance** to implement the validate_email() function based strictly on the behavior described by the test cases.

The implementation should be driven entirely by the test expectations.

Expected Outcome

- Well-defined unit tests using unittest or pytest
- An AI-generated email validation function
- All test cases passing successfully
- Clear alignment between test cases and function behavior

Prompt:-
Follow Test Driven Development (TDD).
First, create a file test_email.py and write unit tests using unittest for a function validate_email(email).
Test valid and invalid emails (missing @, missing domain, missing username, wrong format, spaces, etc.).
Do not write the function yet.
After writing the tests, create email_validator.py and implement validate_email() so that all tests pass.
Follow the cycle: test → fail → implement → pass**.**

lab 8.4 > test_email.py > TestEmailValidation

```python
import unittest
from email_validator import validate_email
class TestEmailValidation(unittest.TestCase):
    def test_valid_emails(self):
        self.assertTrue(validate_email("user@example.com"))
        self.assertTrue(validate_email("john.doe@gmail.com"))
        self.assertTrue(validate_email("abc123@yahoo.in"))

    def test_missing_at_symbol(self):
        self.assertFalse(validate_email("userexample.com"))

    def test_missing_domain(self):
        self.assertFalse(validate_email("user@"))

    def test_missing_username(self):
        self.assertFalse(validate_email("@example.com"))

    def test_missing_dot_in_domain(self):
        self.assertFalse(validate_email("user@examplecom"))

    def test_space_in_email(self):
        self.assertFalse(validate_email("user @example.com"))


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

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
PS C:\AIAC>
```

Explanation:-
We first wrote unit tests to define valid and invalid email formats.Then, based on these tests, the validate_email() function was implemented.After implementation, all test cases passed successfully.This shows correct use of Test Driven Development.

Task 3: Decision Logic Development Using TDD

Scenario
In a grading or evaluation module, a function is required to determine the maximum value among three inputs. Accuracy is essential, as incorrect results could affect downstream decision logic.

Task Description
Using the **TDD methodology**:

1. Write test cases that describe the expected output for different combinations of three numbers.
2. Prompt **GitHub Copilot or Cursor AI** to implement the function logic

based on the written tests.

Avoid writing any logic before test cases are completed.

Expected Outcome

- Comprehensive test cases covering normal and edge cases
- AI-generated function implementation
- Passing test results demonstrating correctness
- Evidence that logic was derived from tests, not assumptions

Prompt:-create a file test_max_three.py and write unit tests using unittest for a function max_of_three(a, b, c).Test positive numbers, negative numbers, equal values, mixed values, and edge cases. After completing the tests, create a separate file max_three.py and implement max_of_three() so that all tests pass.Keep test code and implementation code in their respective files only.

lab 8.4 > test_max_three.py > ...

```python
import unittest
from max_three import max_of_three
class TestMaxOfThree(unittest.TestCase):

    def test_all_positive(self):
        self.assertEqual(max_of_three(3, 7, 5), 7)

    def test_all_negative(self):
        self.assertEqual(max_of_three(-3, -7, -5), -3)

    def test_mixed_values(self):
        self.assertEqual(max_of_three(-2, 4, 1), 4)

    def test_all_equal(self):
        self.assertEqual(max_of_three(5, 5, 5), 5)

    def test_two_equal_max(self):
        self.assertEqual(max_of_three(6, 6, 2), 6)

    def test_decimal_values(self):
        self.assertEqual(max_of_three(2.5, 3.7, 3.1), 3.7)


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

```
......
----------------------------------
Ran 6 tests in 0.001s

OK
```

Explanation:-
The function max_of_three(a, b, c) is used to find the maximum value among three numbers.first, it checks if a is greater than or equal to both b and c.If this condition is true, it returns a as the largest number.If not, it then checks if b is greater than or equal to both a and c.If this is true, it returns b as the largest number.If neither a nor b is the largest, then c must be the greatest value, so the function returns c

Task 4: Shopping Cart Development with AI-Assisted TDD

Scenario
You are building a simple shopping cart module for an e-commerce application. The cart must support adding items, removing items, and calculating the total price accurately.

Task Description
Follow a **test-driven approach**:

1. Write unit tests for each required behavior:
    - Adding an item
    - Removing an item
    - Calculating the total price
2. After defining all tests, use **AI tools** to generate the ShoppingCart class and its methods so that the tests pass.

Focus on behavior-driven testing rather than implementation details.

Expected Outcome

- Unit tests defining expected shopping cart behavior
- AI-generated class implementation
- All tests passing successfully
- Clear demonstration of TDD applied to a class-based design

Prompt:-First, create a file test_cart.py and write unit tests using unittest for a ShoppingCart class.Test adding items, removing items, and calculating total price. After writing tests, create shopping_cart.py and implement the ShoppingCart class so that all tests pass.

Code:-

lab 8.4 > test_cart.py > TestShoppingCart > test_remove_non_existing_item

```python
import unittest
from shopping_cart import ShoppingCart
class TestShoppingCart(unittest.TestCase):
    def setUp(self):
        self.cart = ShoppingCart()

    def test_add_item(self):
        self.cart.add_item("Book", 100)
        self.assertEqual(len(self.cart.items), 1)

    def test_remove_item(self):
        self.cart.add_item("Pen", 20)
        self.cart.remove_item("Pen")
        self.assertEqual(len(self.cart.items), 0)

    def test_total_price(self):
        self.cart.add_item("Book", 100)
        self.cart.add_item("Pen", 20)
        self.assertEqual(self.cart.get_total(), 120)

    def test_remove_non_existing_item(self):
        self.cart.add_item("Book", 100)
        self.cart.remove_item("Pen")
        self.assertEqual(self.cart.get_total(), 100)
if __name__ == "__main__":
    unittest.main()
```

```
....
----------------------------
Ran 4 tests in 0.000s

OK
```

Explanation:-

The unit tests were written first to define the expected behavior of the shopping cart.

These tests checked adding items, removing items, and calculating the total price.

Initially, the tests failed because the ShoppingCart class was not implemented.

After that, the class and its methods were written according to the test requirements.

When the tests were run again, all test cases passed successfully.

This shows that the implementation was developed based on test cases, following the Test Driven Development process.

Task 5: String Validation Module Using TDD

Scenario

You are working on a text-processing module where a function is required to identify whether a given string is a palindrome. The function must handle different cases and inputs reliably.

Task Description

Using **Test Driven Development**:

1. Write test cases for a palindrome checker covering:

- Simple palindromes
- Non-palindromes
- Case variations

2. Use **GitHub Copilot or Cursor AI** to generate the is_palindrome() function based on the test case expectations.

The function should be implemented only after tests are written.

Expected Outcome

- Clearly written test cases defining expected behavior
- AI-assisted implementation of the palindrome checker
- All test cases passing successfully
- Evidence of TDD methodology applied correctly

Prompt:-

First, create a file test_palindrome.py and write unit tests using unittest for a function is_palindrome(text).Test simple palindromes, non-palindromes, and case variations. After writing the tests, create palindrome.py and implement is_palindrome() so that all tests pass..

Code:-

assign-1.4.py | assign-3.4.py | assign-5.4.py | test_squa

lab 8.4 > test_palindrome.py > ...

```python
import unittest
from palindrome import is_palindrome
class TestPalindrome(unittest.TestCase):
    def test_simple_palindrome(self):
        self.assertTrue(is_palindrome("madam"))
        self.assertTrue(is_palindrome("level"))

    def test_not_palindrome(self):
        self.assertFalse(is_palindrome("hello"))
        self.assertFalse(is_palindrome("python"))

    def test_case_variation(self):
        self.assertTrue(is_palindrome("Madam"))
        self.assertTrue(is_palindrome("RaceCar"))

    def test_single_character(self):
        self.assertTrue(is_palindrome("a"))

    def test_empty_string(self):
        self.assertTrue(is_palindrome(""))
if __name__ == "__main__":
    unittest.main()
```

```
.....
----------------------------------------
Ran 5 tests in 0.000s

OK
```

Explanation:-

First, unit tests were written in test_palindrome.py to define the expected behavior of the palindrome checker.These tests verify simple palindromes, non-palindromes, and case variations.Initially, the tests failed because the is_palindrome() function was not implemented.After that, the function was written in palindrome.py according to the test requirements.The function converts the input string to lowercase and compares it with its reverse.If both are equal, it returns True; otherwise, it returns False.After implementation, all test cases passed successfully.

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots