

# **ASSIGNMENT-8**

**NAME:K.Naga Chaitanya**

**HALLTICKET NO:2303A51292**

**BATCH:05**

## **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 Code:**

The screenshot shows two separate instances of the Google Colab interface. Both instances have tabs for 'word - Search', 'Document 11.docx', 'google collab - Search', and 'Untitled30.ipynb - Colab'. The top instance has a status bar showing '9:40 AM' and 'Python 3'. It contains two code cells:

```
[1] 0s
import unittest
# ---- TEST CASES (written first in TDD) ----
class TestSquareFunction(unittest.TestCase):
    def test_positive_number(self):
        self.assertEqual(square(4), 16)
    def test_negative_number(self):
        self.assertEqual(square(-3), 9)
    def test_zero(self):
        self.assertEqual(square(0), 0)
    def test_large_number(self):
        self.assertEqual(square(100), 10000)

[2] 0s
# ---- IMPLEMENTATION (written AFTER tests) ----
def square(n):
    return n * n
```

The bottom instance also has a status bar showing '9:40 AM' and 'Python 3'. It contains three code cells:

```
[1] 0s
def test_positive_number(self):
    self.assertEqual(square(4), 16)
def test_negative_number(self):
    self.assertEqual(square(-3), 9)
def test_zero(self):
    self.assertEqual(square(0), 0)
def test_large_number(self):
    self.assertEqual(square(100), 10000)

[2] 0s
# ---- IMPLEMENTATION (written AFTER tests) ----
def square(n):
    return n * n

[3] 0s
unittest.main(argv=[''], verbosity=2, exit=False)
```

Output:

The screenshot shows a Google Colab interface with a dark theme. The main code editor window displays a Python script named 'Untitled30.ipynb'. The output pane shows the results of a test run:

```
test_large_number (__main__.TestSquareFunction.test_large_number) ... ok
test_negative_number (__main__.TestSquareFunction.test_negative_number) ... ok
test_positive_number (__main__.TestSquareFunction.test_positive_number) ... ok
test_zero (__main__.TestSquareFunction.test_zero) ... ok
-----
Ran 4 tests in 0.009s
OK
<unittest.main.TestProgram at 0x7e0211f2cda0>
```

The status bar at the bottom right indicates the time is 9:40 AM and the Python version is Python 3.

## 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

Code:

```
[4] 0s
import unittest

# ----- TEST CASES (WRITTEN BEFORE FUNCTION) -----
class TestEmailValidation(unittest.TestCase):

    def test_valid_email(self):
        self.assertTrue(validate_email("user@example.com"))

    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_invalid_structure(self):
        self.assertFalse(validate_email("user@com"))

    def test_email_with_numbers(self):
        self.assertTrue(validate_email("user123@gmail.com"))

[5] 0s
#AI-Generated Implementation

import re

# ----- IMPLEMENTATION (AFTER TESTS) -----
def validate_email(email):
    pattern = r'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
    return re.match(pattern, email) is not None

#Run Tests

[6] 0s
unittest.main(argv=[''], verbosity=2, exit=False)
```

Output:

The screenshot shows a Google Colab interface with a terminal window displaying test results. The terminal output includes:

```
test_email_with_numbers (_main_.TestEmailValidation.test_email_with_numbers) ... ok
test_invalid_structure (_main_.TestEmailValidation.test_invalid_structure) ... ok
test_missing_at_symbol (_main_.TestEmailValidation.test_missing_at_symbol) ... ok
test_missing_domain (_main_.TestEmailValidation.test_missing_domain) ... ok
test_missing_username (_main_.TestEmailValidation.test_missing_username) ... ok
test_valid_email (_main_.TestEmailValidation.test_valid_email) ... ok
test_large_number (_main_.TestSquareFunction.test_large_number) ... ok
test_negative_number (_main_.TestSquareFunction.test_negative_number) ... ok
test_positive_number (_main_.TestSquareFunction.test_positive_number) ... ok
test_zero (_main_.TestSquareFunction.test_zero) ... ok
-----
Ran 10 tests in 0.023s
OK
<unittest.main.TestProgram at 0x7e0211f2eab0>
```

### 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

**Code:**

```
[7] 0s
import unittest

# ----- TEST CASES FIRST (TDD) -----
class TestMaxOfThree(unittest.TestCase):

    def test_normal_numbers(self):
        self.assertEqual(max_of_three(2, 8, 5), 8)

    def test_first_is_largest(self):
        self.assertEqual(max_of_three(10, 3, 6), 10)

    def test_negative_numbers(self):
        self.assertEqual(max_of_three(-1, -5, -3), -1)

    def test_all_equal(self):
        self.assertEqual(max_of_three(4, 4, 4), 4)

    def test_two_equal_largest(self):
        self.assertEqual(max_of_three(7, 7, 2), 7)

[8] 0s
def test_all_equal(self):
    self.assertEqual(max_of_three(4, 4, 4), 4)

def test_two_equal_largest(self):
    self.assertEqual(max_of_three(7, 7, 2), 7)

#AI-Generated Implementation

# ----- IMPLEMENTATION (AFTER TESTS) -----
def max_of_three(a, b, c):
    return max(a, b, c)

#Run Tests

[9] 0s
unittest.main(argv=[''], verbosity=2, exit=False)
```

**Output:**

A screenshot of a Google Colab interface. The main window shows a terminal session with the following output:

```
test_email_with_numbers (_main_.TestEmailValidation.test_email_with_numbers) ... ok
test_invalid_structure (_main_.TestEmailValidation.test_invalid_structure) ... ok
test_missing_at_symbol (_main_.TestEmailValidation.test_missing_at_symbol) ... ok
test_missing_domain (_main_.TestEmailValidation.test_missing_domain) ... ok
test_missing_username (_main_.TestEmailValidation.test_missing_username) ... ok
test_valid_email (_main_.TestEmailValidation.test_valid_email) ... ok
test_all_equal (_main_.TestMaxOfThree.test_all_equal) ... ok
test_first_is_largest (_main_.TestMaxOfThree.test_first_is_largest) ... ok
test_negative_numbers (_main_.TestMaxOfThree.test_negative_numbers) ... ok
test_normal_numbers (_main_.TestMaxOfThree.test_normal_numbers) ... ok
test_two_equal_largest (_main_.TestMaxOfThree.test_two_equal_largest) ... ok
test_large_number (_main_.TestSquareFunction.test_large_number) ... ok
test_negative_number (_main_.TestSquareFunction.test_negative_number) ... ok
test_positive_number (_main_.TestSquareFunction.test_positive_number) ... ok
test_zero (_main_.TestSquareFunction.test_zero) ... ok

Ran 15 tests in 0.033s
OK
<unittest.main.TestProgram at 0x7e0211f2d0a0>
```

The terminal window has a dark theme. The status bar at the bottom right shows "9:55 AM" and "Python 3".

## 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:

- o Adding an item
- o Removing an item
- o 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

### CODE:

```
[10] ✓ 0s
import unittest

# ----- TESTS FIRST (TDD RULE) -----
class TestShoppingCart(unittest.TestCase):

    def test_add_item(self):
        cart = ShoppingCart()
        cart.add_item("Book", 100)
        self.assertEqual(cart.calculate_total(), 100)

    def test_add_multiple_items(self):
        cart = ShoppingCart()
        cart.add_item("Book", 100)
        cart.add_item("Pen", 20)
        self.assertEqual(cart.calculate_total(), 120)

    def test_remove_item(self):
        cart = ShoppingCart()
        cart.add_item("Book", 100)
        cart.remove_item("Book")
        self.assertEqual(cart.calculate_total(), 0)

[11] ✓ 0s
# ----- IMPLEMENTATION AFTER TESTS -----
class ShoppingCart:

    def __init__(self):
        self.items = {}

    def add_item(self, name, price):
        self.items[name] = price

    def remove_item(self, name):
        if name in self.items:
            del self.items[name]

    def calculate_total(self):
        return sum(self.items.values())

```

The screenshot shows a Google Colab notebook titled "Untitled30.ipynb". The code editor contains the following Python code:

```

class ShoppingCart:
    def __init__(self):
        self.items = {}

    def add_item(self, name, price):
        self.items[name] = price

    def remove_item(self, name):
        if name in self.items:
            del self.items[name]

    def calculate_total(self):
        return sum(self.items.values())

```

Below the code, there is a section titled "#Run Tests" containing:

```

unittest.main(argv=[''], verbosity=2, exit=False)

```

The status bar at the bottom indicates "Variables" and "Terminal". A floating window titled "Snipping Tool" shows a message: "Screenshot copied to clipboard" and "Automatically saved to screenshots folder".

## Output:

The screenshot shows the execution output of the unit tests in the Google Colab notebook. The output is as follows:

```

test_invalid_structure (_main_.TestEmailValidation.test_invalid_structure) ... ok
test_missing_at_symbol (_main_.TestEmailValidation.test_missing_at_symbol) ... ok
test_missing_domain (_main_.TestEmailValidation.test_missing_domain) ... ok
test_missing_username (_main_.TestEmailValidation.test_missing_username) ... ok
test_valid_email (_main_.TestEmailValidation.test_valid_email) ... ok
test_all_equal (_main_.TestMaxOfThree.test_all_equal) ... ok
test_first_is_largest (_main_.TestMaxOfThree.test_first_is_largest) ... ok
test_negative_numbers (_main_.TestMaxOfThree.test_negative_numbers) ... ok
test_normal_numbers (_main_.TestMaxOfThree.test_normal_numbers) ... ok
test_two_equal_largest (_main_.TestMaxOfThree.test_two_equal_largest) ... ok
test_add_item (_main_.TestShoppingCart.test_add_item) ... ok
test_add_multiple_items (_main_.TestShoppingCart.test_add_multiple_items) ... ok
test_remove_item (_main_.TestShoppingCart.test_remove_item) ... ok
test_remove_non_existing_item (_main_.TestShoppingCart.test_remove_non_existing_item) ... ok
test_large_number (_main_.TestSquareFunction.test_large_number) ... ok
test_negative_number (_main_.TestSquareFunction.test_negative_number) ... ok
test_positive_number (_main_.TestSquareFunction.test_positive_number) ... ok
test_zero (_main_.TestSquareFunction.test_zero) ... ok

-----
Ran 19 tests in 0.029s
OK
<unittest.main.TestProgram at 0x7e0211f2d700>

```

The status bar at the bottom indicates "Variables", "Terminal", "9:58 AM", and "Python 3".

## 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:

- o Simple palindromes
- o Non-palindromes o

##### 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

**CODE:**

The screenshot shows a Google Colab interface with a Jupyter notebook titled "Untitled30.ipynb". The code in cell [13] defines a test class `TestPalindrome` that contains six test methods: `test\_simple\_palindrome`, `test\_not\_palindrome`, `test\_case\_insensitive`, `test\_with\_spaces`, `test\_with\_punctuation`, and `test\_single\_character`. Each method uses `self.assertTrue` to check if the `is\_palindrome` function returns `True` for its respective input string.

```
import unittest

# ----- TEST CASES FIRST (TDD) -----
class TestPalindrome(unittest.TestCase):

    def test_simple_palindrome(self):
        self.assertTrue(is_palindrome("madam"))

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

    def test_case_insensitive(self):
        self.assertTrue(is_palindrome("Madam"))

    def test_with_spaces(self):
        self.assertTrue(is_palindrome("nurses run"))

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

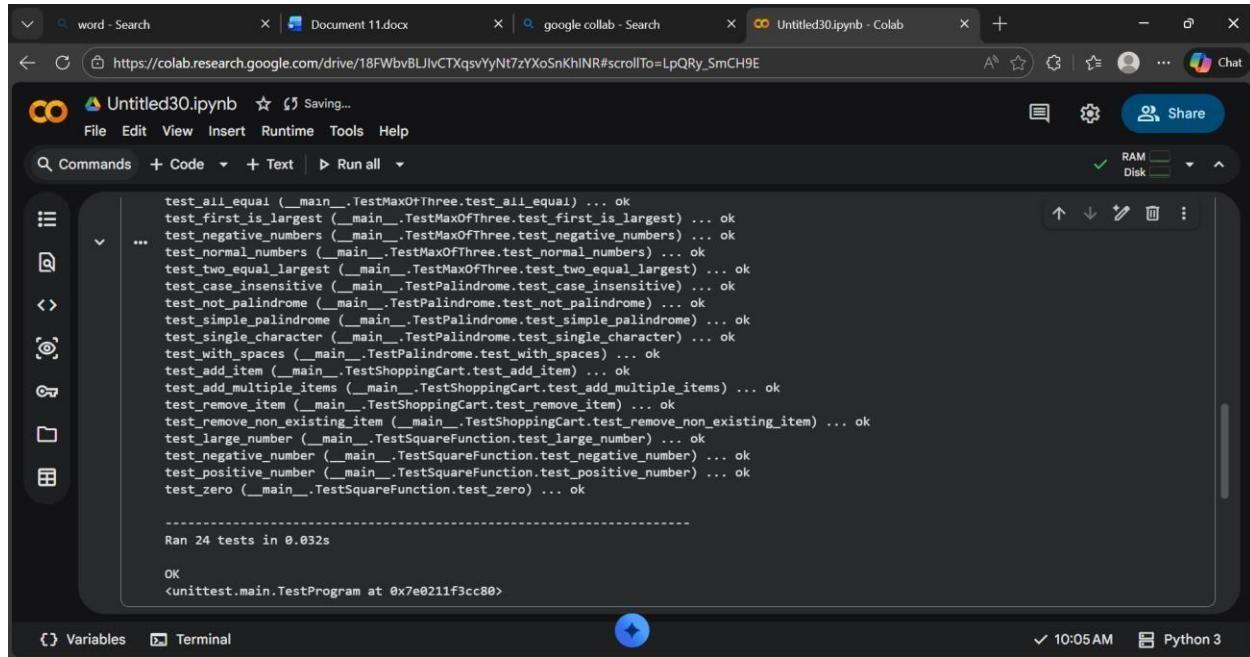
    def test_single_character(self):
        self.assertTrue(is_palindrome("a"))
```

The screenshot shows the continuation of the Google Colab session. In cell [14], under the heading "#AI Implemented Code", an AI-generated implementation of the `is\_palindrome` function is provided. It replaces spaces with nothing, converts the string to lowercase, and then checks if it is equal to its reverse. In cell [15], under the heading "#Run Tests", the command `unittest.main(argv=[''], verbosity=2, exit=False)` is run to execute the tests defined in the notebook.

```
def is_palindrome(s):
    s = s.replace(" ", "").lower()
    return s == s[::-1]
```

```
unittest.main(argv=[''], verbosity=2, exit=False)
```

## Output:



The screenshot shows a Google Colab interface with multiple tabs at the top: "word - Search", "Document 11.docx", "google colab - Search", and "Untitled30.ipynb - Colab". The main content area displays the output of a Python script, specifically the results of a unit test run. The output is as follows:

```
test_all_equal (_main_.TestMaxOfThree.test_all_equal) ... ok
test_first_is_largest (_main_.TestMaxOfThree.test_first_is_largest) ... ok
test_negative_numbers (_main_.TestMaxOfThree.test_negative_numbers) ... ok
test_normal_numbers (_main_.TestMaxOfThree.test_normal_numbers) ... ok
test_two_equal_largest (_main_.TestMaxOfThree.test_two_equal_largest) ... ok
test_case_insensitive (_main_.TestPalindrome.test_case_insensitive) ... ok
test_not_palindrome (_main_.TestPalindrome.test_not_palindrome) ... ok
test_simple_palindrome (_main_.TestPalindrome.test_simple_palindrome) ... ok
test_single_character (_main_.TestPalindrome.test_single_character) ... ok
test_with_spaces (_main_.TestPalindrome.test_with_spaces) ... ok
test_add_item (_main_.TestShoppingCart.test_add_item) ... ok
test_add_multiple_items (_main_.TestShoppingCart.test_add_multiple_items) ... ok
test_remove_item (_main_.TestShoppingCart.test_remove_item) ... ok
test_remove_non_existing_item (_main_.TestShoppingCart.test_remove_non_existing_item) ... ok
test_large_number (_main_.TestSquareFunction.test_large_number) ... ok
test_negative_number (_main_.TestSquareFunction.test_negative_number) ... ok
test_positive_number (_main_.TestSquareFunction.test_positive_number) ... ok
test_zero (_main_.TestSquareFunction.test_zero) ... ok

-----
Ran 24 tests in 0.032s
OK
<unittest.main.TestProgram at 0x7e021f3cc80>
```

The bottom status bar indicates the time as "10:05 AM" and the Python version as "Python 3".