

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 •

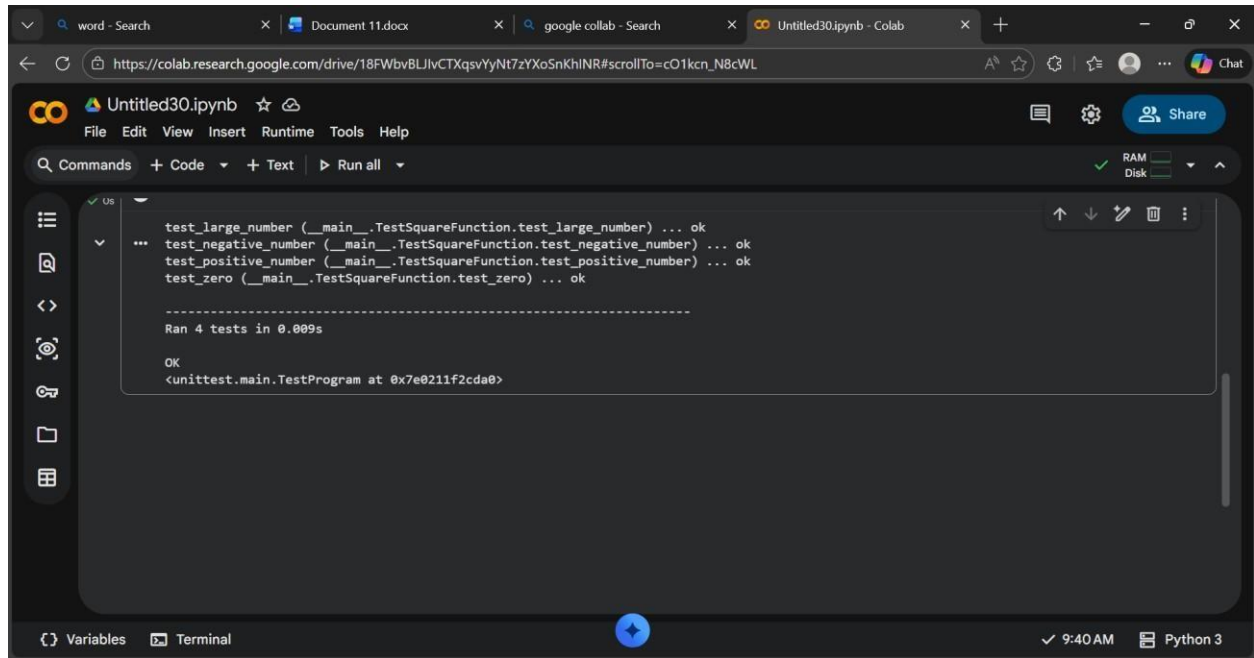
Demonstration of the TDD cycle: test → fail → implement → pass Code:

The image displays two sequential screenshots of a Google Colab notebook titled 'Untitled30.ipynb'. The browser address bar shows the URL: https://colab.research.google.com/drive/18FWbv8LJlvCTXqsvYyNt7zYXoSnKhINR#scrollTo=cO1kcN_N8cWL.

Top Screenshot: The notebook contains two code cells. Cell [1] defines a test class `TestSquareFunction` with four test methods: `test_positive_number`, `test_negative_number`, `test_zero`, and `test_large_number`. Cell [2] implements the `square` function, which returns `n * n`. The status bar at the bottom indicates 'Python 3' and the time '9:40 AM'.

Bottom Screenshot: The notebook has been updated with a third code cell. Cell [3] adds the line `unittest.main(argv=[''], verbosity=2, exit=False)` to run the tests. The status bar remains the same.

Output:



The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The code cell contains a unittest test suite for a 'TestSquareFunction' class. The tests include 'test_large_number', 'test_negative_number', 'test_positive_number', and 'test_zero', all of which passed. The output shows 'Ran 4 tests in 0.009s' and 'OK'. The bottom status bar indicates 'Python 3' and '9:40 AM'.

```
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>
```

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.

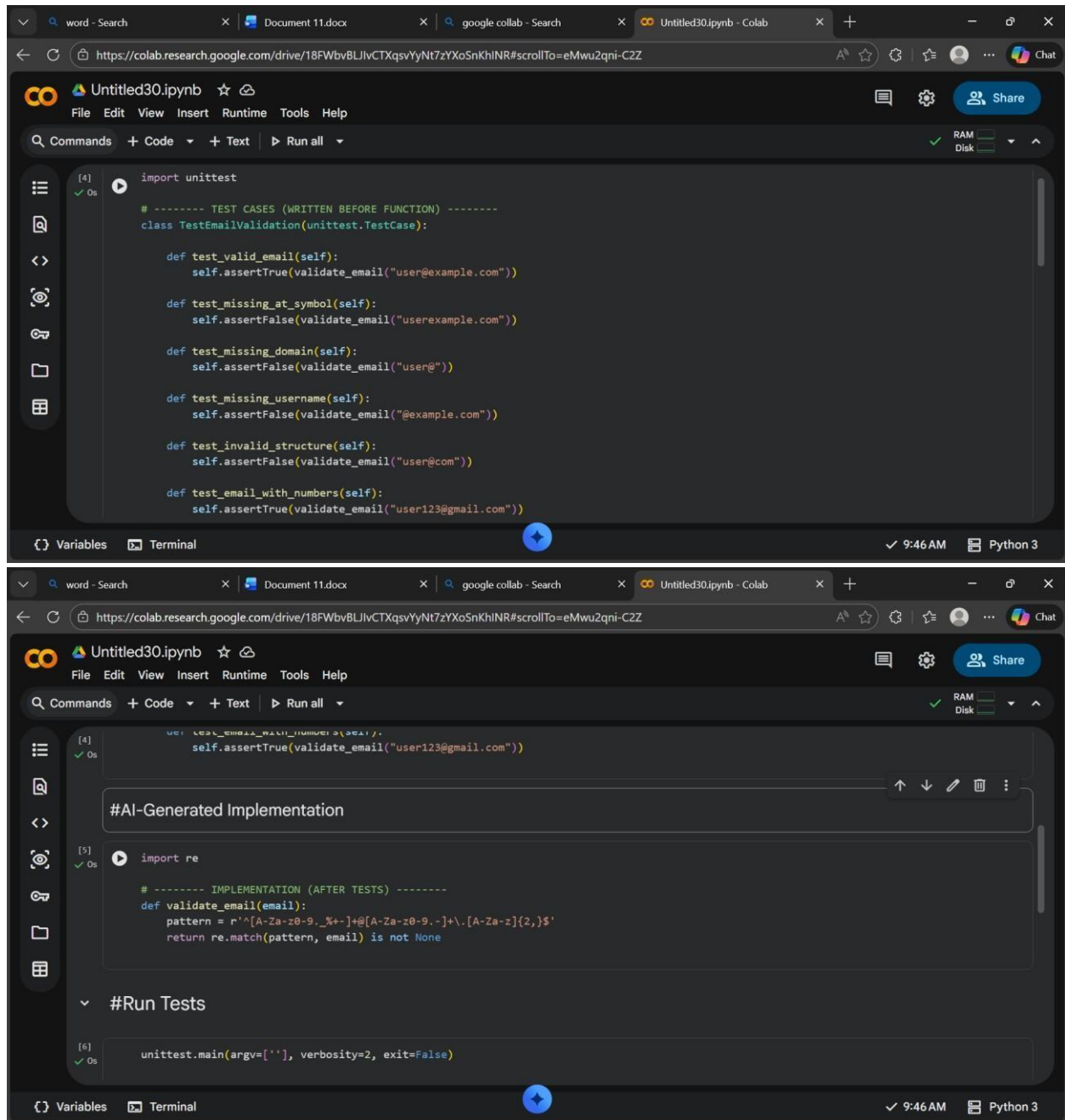
Expected Outcome

- 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:



The image displays two screenshots of a Google Colab notebook titled "Untitled30.ipynb".

Top Screenshot: The notebook shows a code cell [4] with the following Python code:

```
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"))
```

Bottom Screenshot: The notebook shows the implementation of the `validate_email` function and the execution of the tests.

Cell [5] contains the implementation:

```
import re

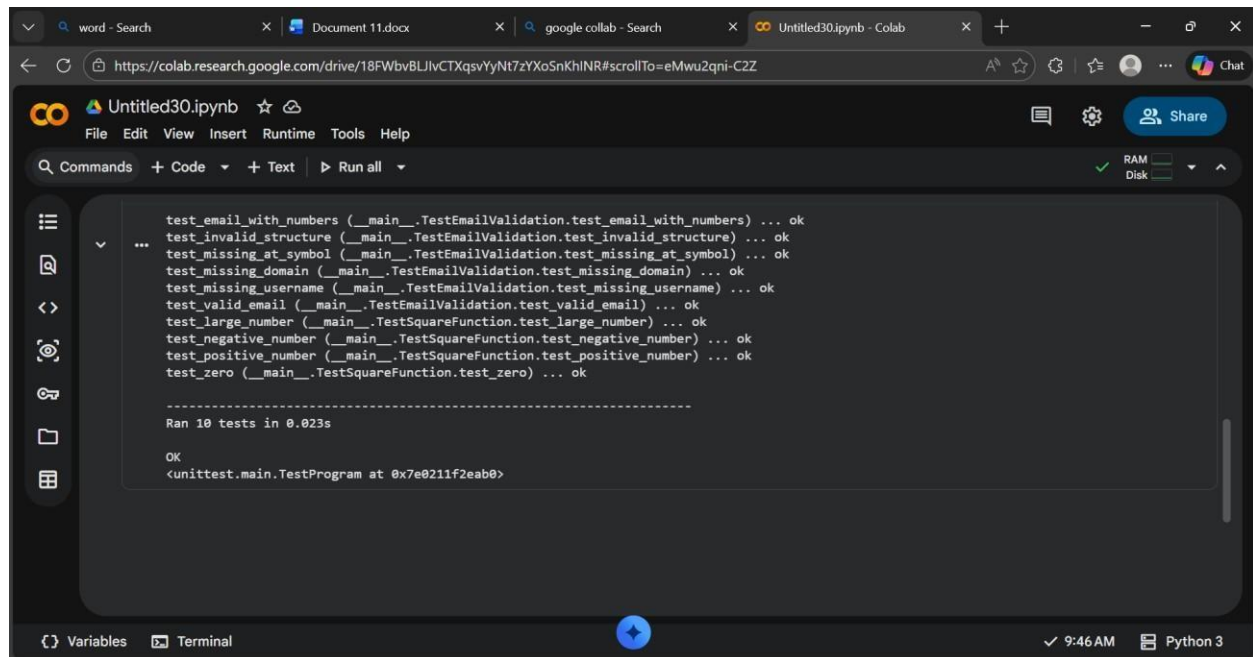
# ----- IMPLEMENTATION (AFTER TESTS) -----
def validate_email(email):
    pattern = r'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
    return re.match(pattern, email) is not None
```

Cell [6] contains the test execution command:

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

Output:

•



The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The notebook contains a series of test cases for email validation and a square function. The tests are as follows:

```
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
```

The tests ran successfully, with a total of 10 tests completed in 0.023 seconds. The output shows 'OK' and the location of the test program 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:

The image displays two screenshots of a Google Colab notebook titled 'Untitled30.ipynb'. The notebook is open in a web browser with the URL https://colab.research.google.com/drive/18FWbvBLJlvCTXqsvYyNt7zYXoSnKhINR#scrollTo=9M7Qje_F_4ay.

Top Screenshot: The notebook shows the initial test cases for a function named `max_of_three`. The code is as follows:

```
[7] 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)
```

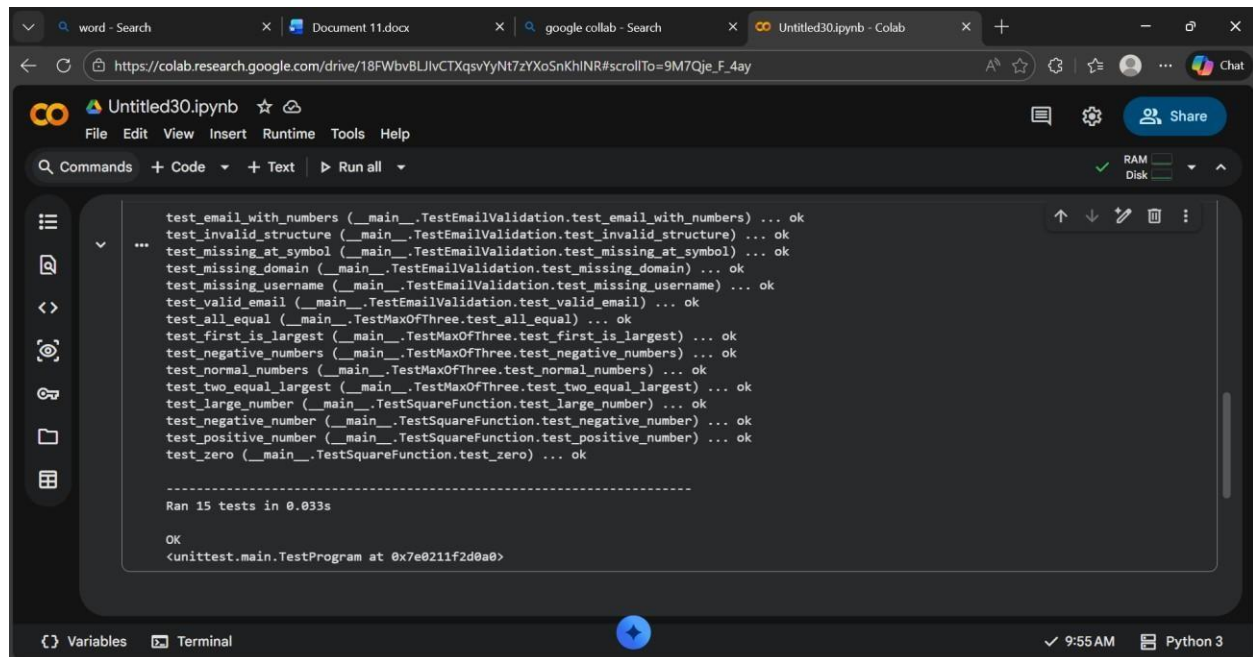
Bottom Screenshot: The notebook shows the implementation of the `max_of_three` function after the tests are written. The code is as follows:

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

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

Output:

•



The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The notebook contains 15 unit tests that have all passed successfully. The tests are organized into two groups: email validation tests and mathematical function tests. The output shows 'Ran 15 tests in 0.033s' and 'OK'.

```
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>
```

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:

The image displays two screenshots of a Google Colab notebook, illustrating the Test-Driven Development (TDD) process for a ShoppingCart class.

Top Screenshot: Test Suite

```
[10] ✓ Os
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)
```

Bottom Screenshot: AI-Generated Implementation

```
[11] ✓ Os
# ----- IMPLEMENTATION AFTER TESTS -----
class ShoppingCart:

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

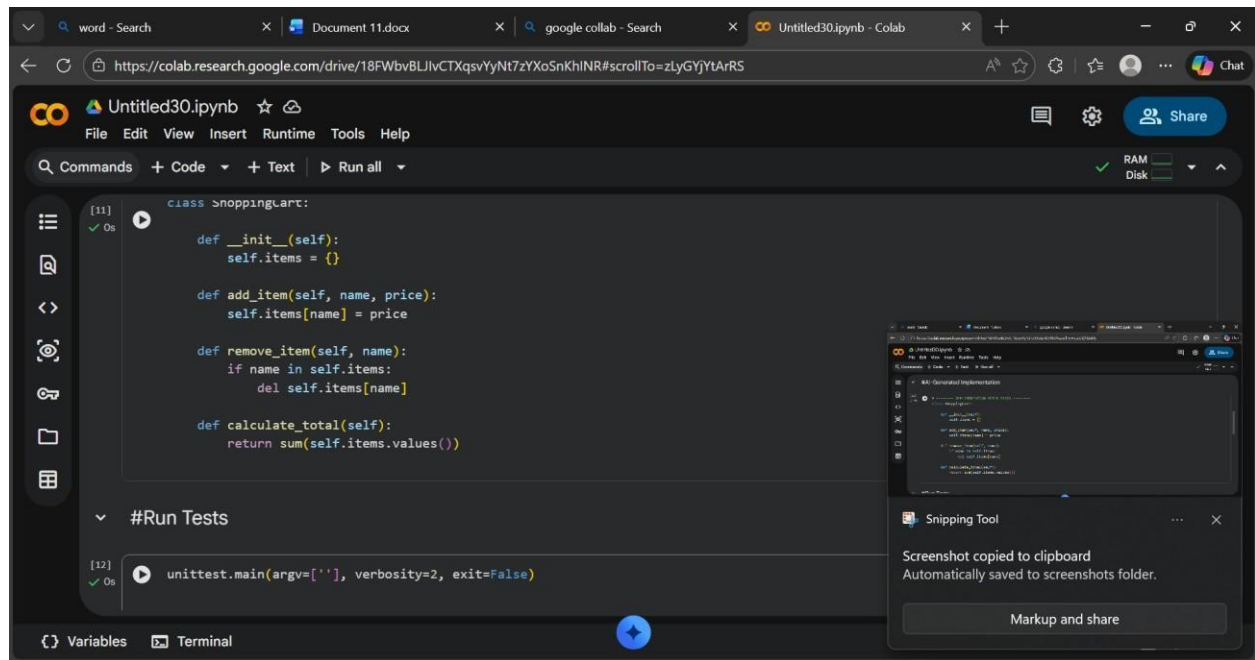
    def add_item(self, name, price):
        self.items[name] = price

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

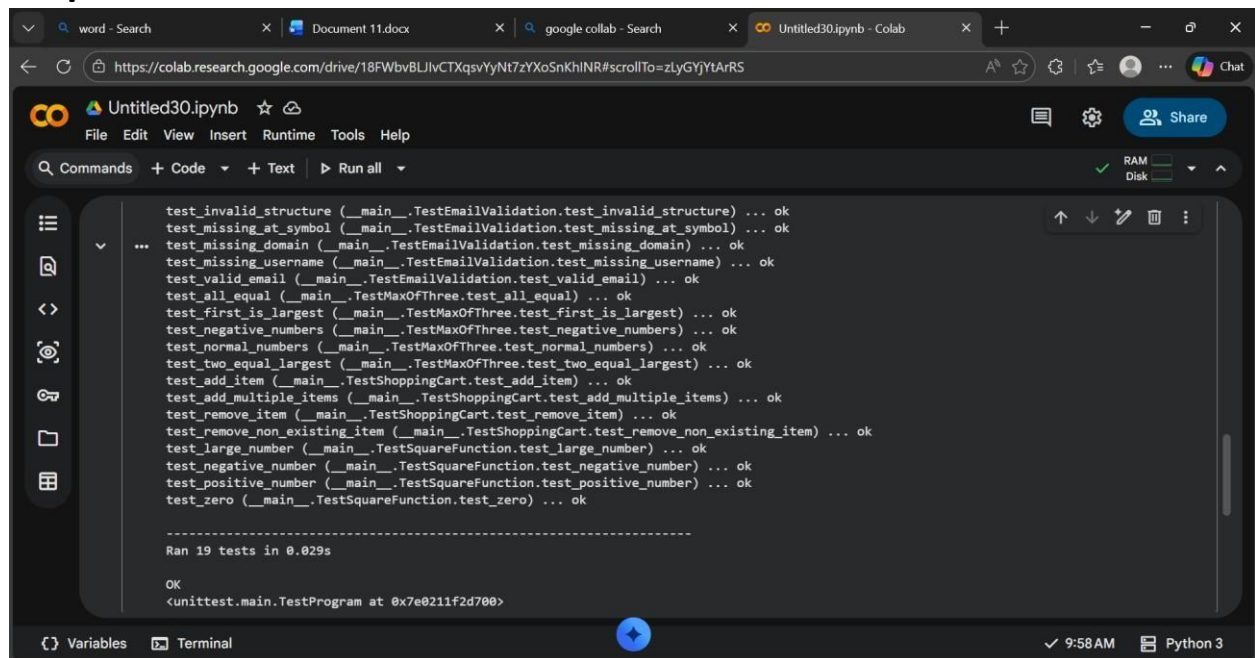
    def calculate_total(self):
        return sum(self.items.values())

# Run Tests
```

Both screenshots show the notebook interface with the file 'Untitled30.ipynb' and the Python 3 runtime environment. The status bar at the bottom indicates the time as 9:58 AM.



Output:



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:

word - Search | Document 11.docx | google colab - Search | Untitled30.ipynb - Colab

https://colab.research.google.com/drive/18FWbvBLJlvCTXqsvYyNt7zYXoSnKHiNR#scrollTo=Lg3zxpB1dN

Untitled30.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

RAM Disk

```
[13] 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_single_character(self):
        self.assertTrue(is_palindrome("a"))
```

{ } Variables Terminal 10:03 AM Python 3

word - Search | Document 11.docx | google colab - Search | Untitled30.ipynb - Colab

https://colab.research.google.com/drive/18FWbvBLJlvCTXqsvYyNt7zYXoSnKHiNR#scrollTo=LpQRy_SmCH9E

Untitled30.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

RAM Disk

```
[13] self.assertTrue(is_palindrome("nurses run"))

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

#Ai Implemented Code

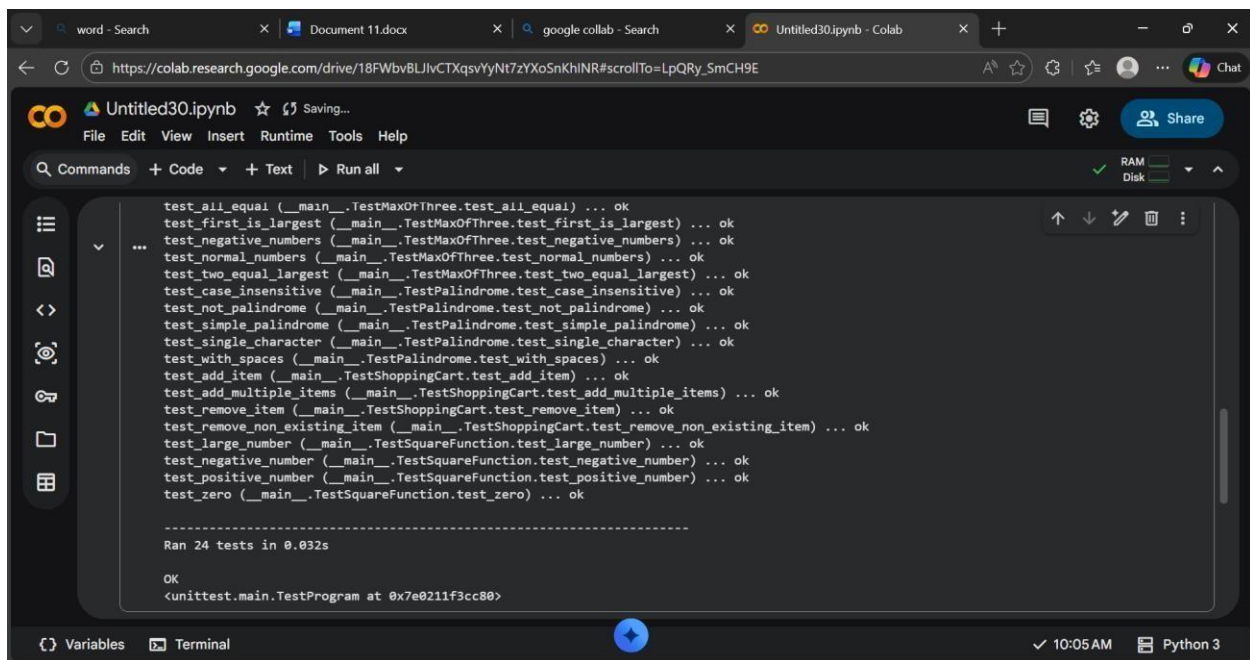
```
[14] # ----- IMPLEMENTATION AFTER TESTS -----
def is_palindrome(s):
    s = s.replace(" ", "").lower()
    return s == s[::-1]
```

#Run Tests

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

{ } Variables Terminal 10:05 AM Python 3

Output:



The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The notebook interface includes a top bar with tabs for 'word - Search', 'Document 11.docx', 'google colab - Search', and 'Untitled30.ipynb - Colab'. The main area displays a list of 24 test cases, all of which passed with the result '... ok'. The tests are organized into three groups: 'TestMaxOfThree', 'TestPalindrome', and 'TestShoppingCart'. Below the list of tests, the output shows 'Ran 24 tests in 0.032s' and 'OK'. The bottom of the notebook shows a 'Variables' tab and a 'Terminal' tab, both of which are currently empty. The status bar at the bottom indicates the time is 10:05 AM and the environment is Python 3.

```
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 0x7e0211f3cc80>
```