

## Assignment 8.4 Ai Assisted Coding

Htno:2303A510C0

Btno:06

### 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 tabs at the top include 'word - Search', 'Document 11.docx', 'google collab - Search', and 'Untitled30.ipynb - Colab'. The address bar shows the URL: [https://colab.research.google.com/drive/18FWbvBLJvCTXqsvYyNt7zYXoSnKHiNR#scrollTo=cO1kcN\\_N8cWL](https://colab.research.google.com/drive/18FWbvBLJvCTXqsvYyNt7zYXoSnKHiNR#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] defines the `square` function implementation. The status bar at the bottom indicates '9:40 AM' and 'Python 3'.

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

**Bottom Screenshot:** The notebook has been updated with a third code cell. Cell [1] now only contains the test methods. Cell [2] contains the `square` function implementation. Cell [3] adds `unittest.main` to execute the tests. The status bar remains the same.

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

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

Output:

```
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'. The top screenshot shows the initial test cases for an email validation function. The bottom screenshot shows the implementation of the function and the execution of the tests.

**Top Screenshot: Test Cases**

```
[4] 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: Implementation and Test Execution**

**#AI-Generated Implementation**

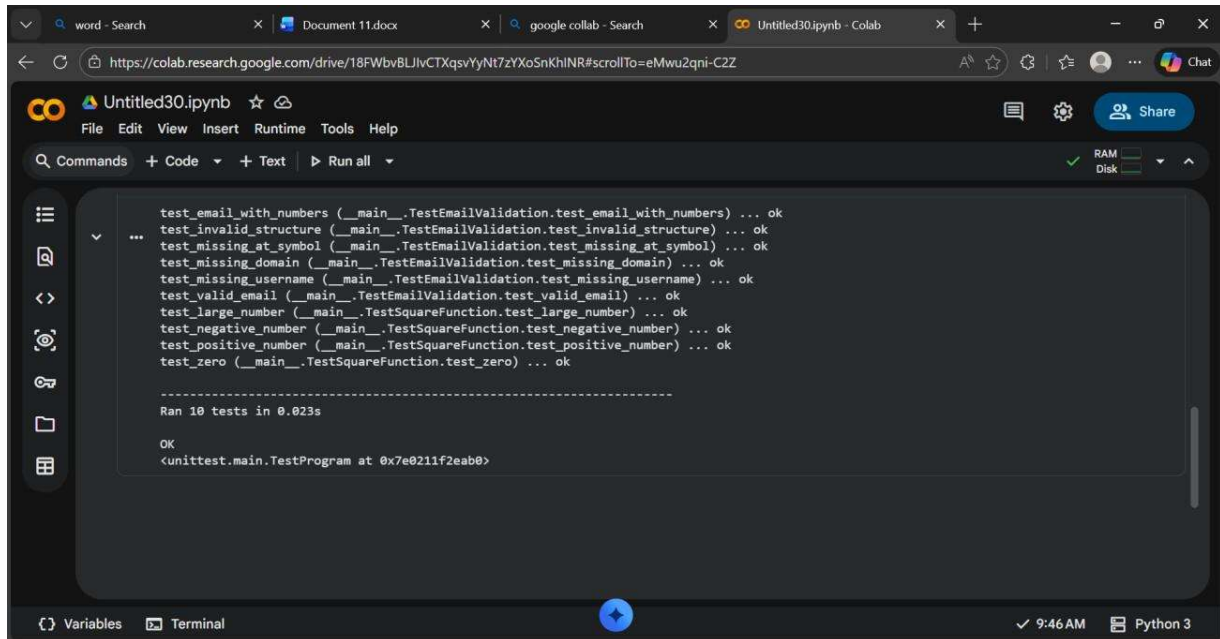
```
[5] import re

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

Output:



The screenshot shows a Google Colab notebook interface. The top bar includes tabs for 'word - Search', 'Document 11.docx', 'google colab - Search', and 'Untitled30.ipynb - Colab'. The address bar shows the URL: <https://colab.research.google.com/drive/18FWbvBLJvCTXqsvYyNt7zYXoSnKhINR#scrollTo=eMwu2qni-C2Z>. The notebook title is 'Untitled30.ipynb'. The left sidebar contains icons for file explorer, search, and other tools. The main area displays the following test results:

```
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>
```

The bottom status bar shows 'Variables', 'Terminal', '9:46 AM', and 'Python 3'.

## 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, illustrating the process of deriving logic from tests.

**Top Screenshot:** The notebook is titled "Untitled30.ipynb". The code cell [7] contains the following Python code:

```
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 function and the execution of the tests. The code cell [8] contains the following Python code:

```
#AI-Generated Implementation

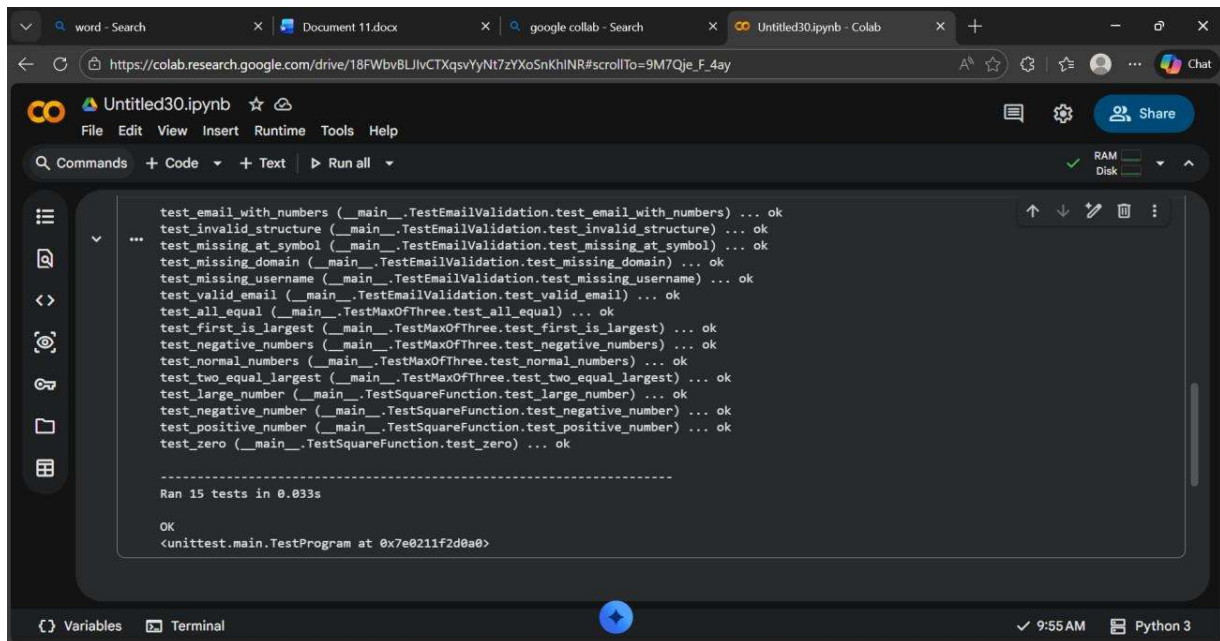
# ----- IMPLEMENTATION (AFTER TESTS) -----
def max_of_three(a, b, c):
    return max(a, b, c)
```

The code cell [9] contains the following Python code:

```
#Run Tests

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

Output:



The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The code cell contains 15 unit tests for two classes: `TestEmailValidation` and `TestMaxOfThree`. All tests pass, indicated by '... ok' at the end of each line. The tests include:

- `test_email_with_numbers`
- `test_invalid_structure`
- `test_missing_at_symbol`
- `test_missing_domain`
- `test_missing_username`
- `test_valid_email`
- `test_all_equal`
- `test_first_is_largest`
- `test_negative_numbers`
- `test_normal_numbers`
- `test_two_equal_largest`
- `test_large_number`
- `test_negative_number`
- `test_positive_number`
- `test_zero`

The output shows 'Ran 15 tests in 0.033s' and 'OK'.

## 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 titled 'Untitled30.ipynb', illustrating the Test-Driven Development (TDD) process for a ShoppingCart class.

**Top Screenshot: Test Cases (TDD Rule)**

```

[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: Implementation After Tests**

```

[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())
  
```

The bottom screenshot also shows a section titled '#Run Tests' at the bottom of the code cell.



The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The code in the first cell defines a `ShoppingCart` class with methods `__init__`, `add_item`, `remove_item`, and `calculate_total`. The second cell, labeled '#Run Tests', contains `unittest.main(argv=[''], verbosity=2, exit=False)`. A 'Snipping Tool' notification is visible on the right, stating 'Screenshot copied to clipboard. Automatically saved to screenshots folder.' The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for file operations, and a status bar at the bottom showing 'Variables' and 'Terminal'.

Output:

The screenshot shows the output of the `unittest.main` command from the previous cell. The output lists 19 tests, all of which passed (ok). The tests include `test_invalid_structure`, `test_missing_at_symbol`, `test_missing_domain`, `test_missing_username`, `test_valid_email`, `test_all_equal`, `test_first_is_largest`, `test_negative_numbers`, `test_normal_numbers`, `test_two_equal_largest`, `test_add_item`, `test_add_multiple_items`, `test_remove_item`, `test_remove_non_existing_item`, `test_large_number`, `test_negative_number`, `test_positive_number`, and `test_zero`. The output concludes with 'Ran 19 tests in 0.029s' and 'OK'. The status bar at the bottom shows '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 image displays two sequential screenshots of a Google Colab notebook titled "Untitled30.ipynb".

**Top Screenshot:** The notebook is at cell [13]. The code defines a `TestPalindrome` class inheriting from `unittest.TestCase`. It includes five test methods: `test_simple_palindrome` (checking "madam"), `test_not_palindrome` (checking "hello"), `test_case_insensitive` (checking "Madam"), `test_with_spaces` (checking "nurses run"), and `test_single_character` (checking "a"). The status bar shows "10:03 AM" and "Python 3".

**Bottom Screenshot:** The notebook is at cell [15]. Cell [13] is collapsed. Cell [14], titled "#Ai Implemented Code", contains the implementation of the `is_palindrome` function:   

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

  
Cell [15], titled "#Run Tests", contains the command `unittest.main(argv=[''], verbosity=2, exit=False)` to execute the tests. The status bar shows "10:05 AM" and "Python 3".

Output:

word - Search x Document 11.docx x google collab - Search x Untitled30.ipynb - Colab x +

https://colab.research.google.com/drive/18FWbvBLJvCTXqsvYyNt7zYXoSnKhINR#scrollTo=LpQRy\_5mCH9E

Untitled30.ipynb ☆ Saving... File Edit View Insert Runtime Tools Help

Commands + Code + Text ▶ Run all

RAM Disk

```
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>
```

Variables Terminal 10:05 AM Python 3