

Assignment 8.4 Ai Assisted Coding

Htno:2303a510C3

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 shows two screenshots of a Google Colab notebook titled "Untitled30.ipynb".

Screenshot 1:

- Cell 1:** Contains Python test code using the `unittest` module. It defines a class `TestSquareFunction` with four test methods: `test_positive_number`, `test_negative_number`, `test_zero`, and `test_large_number`. Each method uses `self.assertEqual` to check if the square of a given number equals the expected result.
- Cell 2:** Contains the implementation of the `square` function, which returns the square of its input parameter `n`.

Screenshot 2:

- Cell 1:** Same as Cell 1 in Screenshot 1.
- Cell 2:** Same as Cell 2 in Screenshot 1.
- Cell 3:** Contains the command `unittest.main(argv=[''], verbosity=2, exit=False)` to run the tests.

The interface includes tabs for "Variables" and "Terminal" at the bottom, and a status bar showing "9:40 AM" and "Python 3".

Output:

The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The code cell contains a Python test script for a 'TestSquareFunction' class. The output shows four tests: 'test_large_number', 'test_negative_number', 'test_positive_number', and 'test_zero', all passing ('ok'). The total execution time was 0.009 seconds. The status bar at the bottom right indicates it's a Python 3 environment at 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:

```

[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"))

[5] #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

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

```

The screenshot shows a Google Colab interface with two code cells. Cell [4] contains a series of test cases for validating email addresses using the `unittest` module. Cell [5] contains the AI-generated implementation of the `validate_email` function, which uses regular expressions to check the email structure. Cell [6] runs the tests, although no output is visible in the screenshot.

Output:

The screenshot shows a Google Colab interface with a dark theme. The main content area displays the output of a Python script, specifically the results of a unit test run. The output includes several test cases for email validation and square function tests, all of which passed ('ok'). A summary at the bottom indicates 18 tests ran in 0.023 seconds and ends with '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_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 18 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.

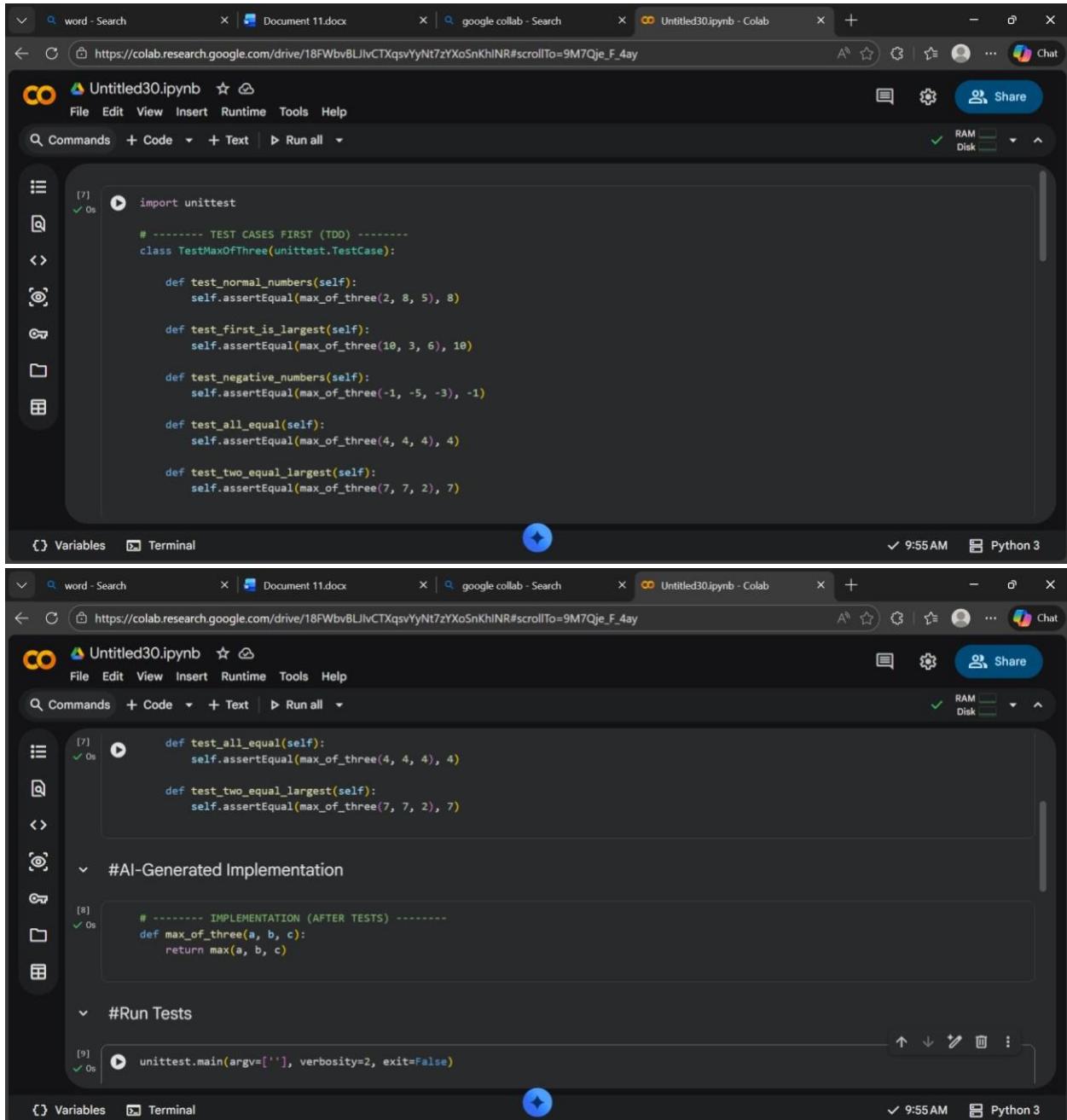
Avoid writing any logic before test cases are completed.

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:



The screenshot shows two versions of a Python script in Google Colab, labeled [7] and [8].

```
[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)

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

The code in [7] contains seven test cases for the `max_of_three` function. The code in [8] is the AI-generated implementation of the `max_of_three` function, which simply returns the maximum value of three arguments using Python's built-in `max` function.

Output:

A screenshot of a Google Colab interface. The top navigation bar shows tabs for 'word - Search', 'Document 11.docx', 'google colab - Search', and 'Untitled30.ipynb - Colab'. The main area is a terminal window titled 'Untitled30.ipynb'. It displays a series of test cases from a unit test suite, all of which pass ('ok'). The tests include various validation functions like 'test_email_with_numbers', 'test_invalid_structure', 'test_missing_at_symbol', etc. At the bottom of the terminal output, it says 'Ran 15 tests in 0.033s' and 'OK <unittest.main.TestProgram at 0x7e0211f2d0a0>'. The Colab interface includes a sidebar with file navigation icons and a status bar at the bottom indicating 'Variables', 'Terminal', '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.

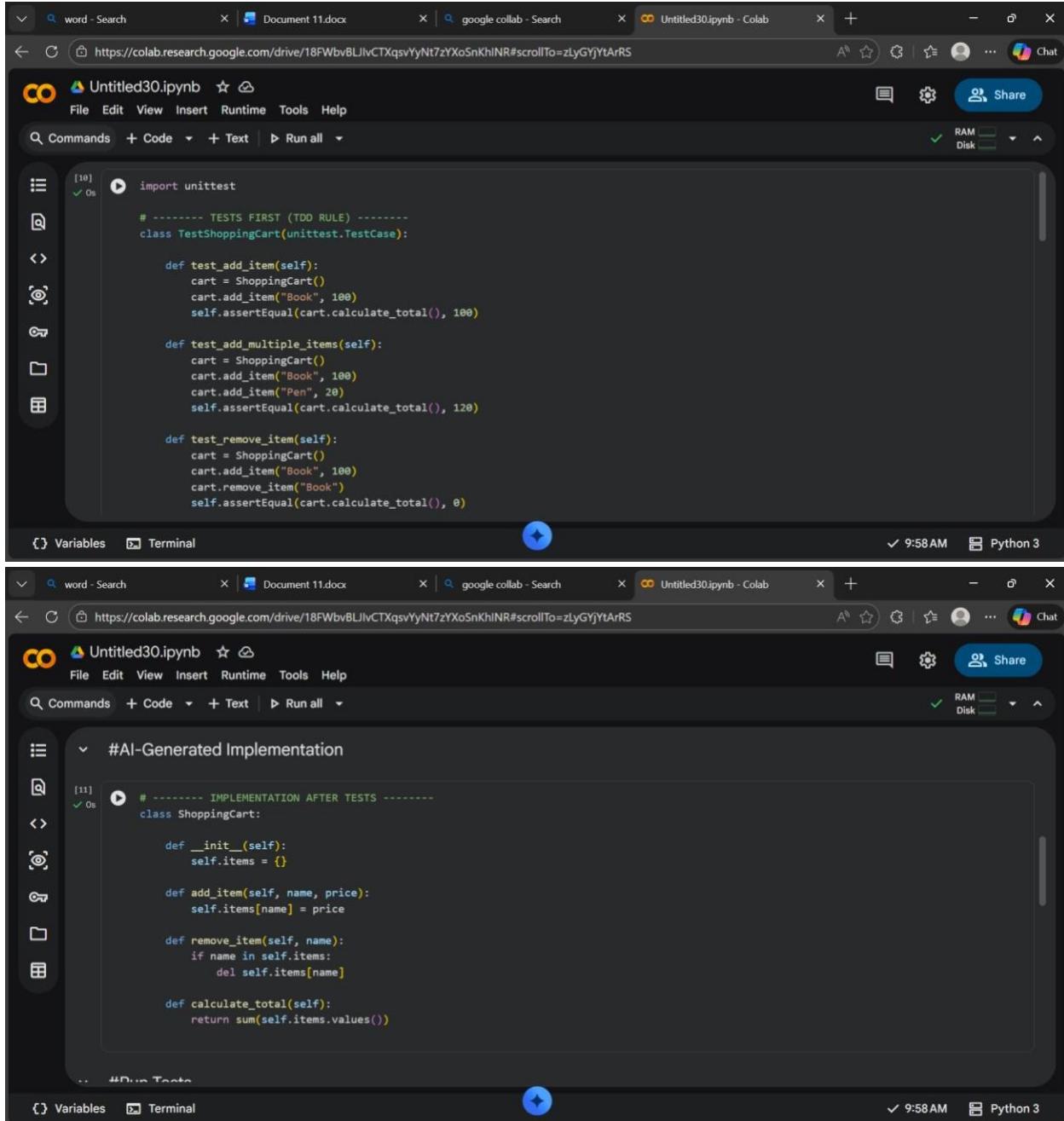
Focus on behavior-driven testing rather than implementation details.

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] word - Search | Document 11.docx | google colab - Search | Untitled30.ipynb - Colab
https://colab.research.google.com/drive/18FWbvBLJlvCTXqsvYyNt7zYXoSnhINR#scrollTo=zLyGYjYtArRS
Untitled30.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
RAM Disk
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] word - Search | Document 11.docx | google colab - Search | Untitled30.ipynb - Colab
https://colab.research.google.com/drive/18FWbvBLJlvCTXqsvYyNt7zYXoSnhINR#scrollTo=zLyGYjYtArRS
Untitled30.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
RAM Disk
#AI-Generated Implementation

# ----- 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())

```

```

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

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

Output:

```

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>

```

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 two consecutive screenshots of a Google Colab notebook titled "Untitled30.ipynb".

Screenshot 1: The code defines a test class `TestPalindrome` with six test methods. The first four tests check for palindromes with different cases ("madam", "hello", "Madam", "nurses run"). The last two tests check for single characters ("a"). All test assertions are failing.

```
[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"))
```

Screenshot 2: The code has been implemented. The first four test methods now pass. A new section titled "#Ai Implemented Code" contains the implementation of the `is_palindrome` function. The final test for a single character also passes.

```
[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)
```

Output:

A screenshot of a Google Colab interface. The top navigation bar shows tabs for 'word - Search', 'Document 11.docx', 'google colab - Search', and 'Untitled30.ipynb - Colab'. The main area is a terminal window titled 'Untitled30.ipynb' with the status 'Saving...'. The terminal output shows the results of a unit test run:

```
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>
```

The bottom right corner of the terminal window shows the time '10:05 AM' and the Python version 'Python 3'. The left sidebar of the Colab interface is visible, showing various notebooks and files.