

## Assignment 8.4 Ai Assisted Coding

Htno:2303a51983

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 colab - Search', and 'Untitled30.ipynb - Colab'. The notebook interface shows a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu is a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all'. The left sidebar contains icons for file management and a 'Variables' panel. The main code area is divided into two cells. Cell [1] contains unit tests for a square function, and Cell [2] contains the implementation of the square function. The status bar at the bottom indicates '9:40 AM' and 'Python 3'.

```
[1] import unittest
# ---- TEST CASES (written first in TOD) ----
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)

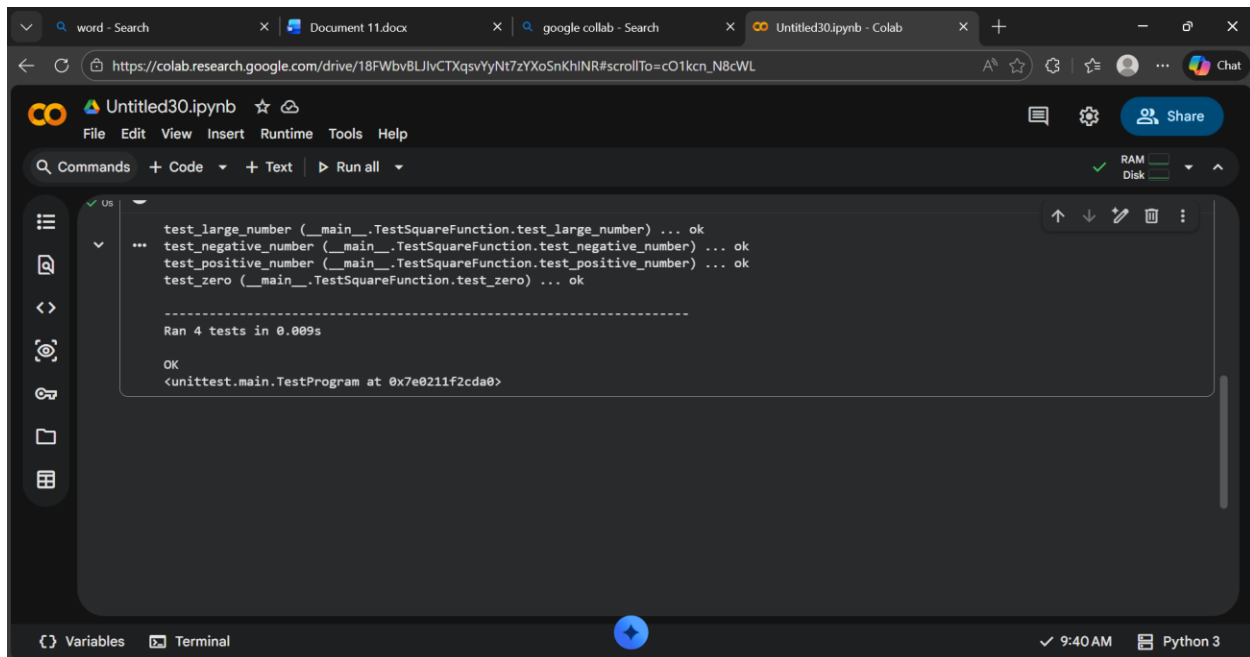
    def test_large_number(self):
        self.assertEqual(square(100), 10000)

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

The second screenshot shows the same notebook with an additional cell [3] at the bottom. Cell [3] contains the command to run the unit tests. The status bar still shows '9:40 AM' and 'Python 3'.

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

Output:



The screenshot shows a Google Colab notebook interface. The browser tabs at the top include 'word - Search', 'Document 11.docx', 'google colab - Search', and 'Untitled30.ipynb - Colab'. The notebook's address bar shows a Google Drive link. The notebook title is 'Untitled30.ipynb'. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu is a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all'. On the right, there are icons for chat, settings, and a 'Share' button, along with RAM and Disk usage indicators. The left sidebar contains icons for file explorer, search, and other notebook functions. The main code cell shows a Python test suite using unittest. The output indicates that four tests passed successfully in 0.009 seconds. The bottom status bar shows 'Variables', 'Terminal', a refresh icon, the time '9:40 AM', and 'Python 3'.

```
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 shows two screenshots of a Google Colab notebook, illustrating the 'Test-Driven Development' (TDD) process. The top screenshot shows the initial state where test cases are written before the function implementation. The bottom screenshot shows the implementation of the function and the execution of the tests.

**Top Screenshot: Test Cases Written Before Function**

```
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

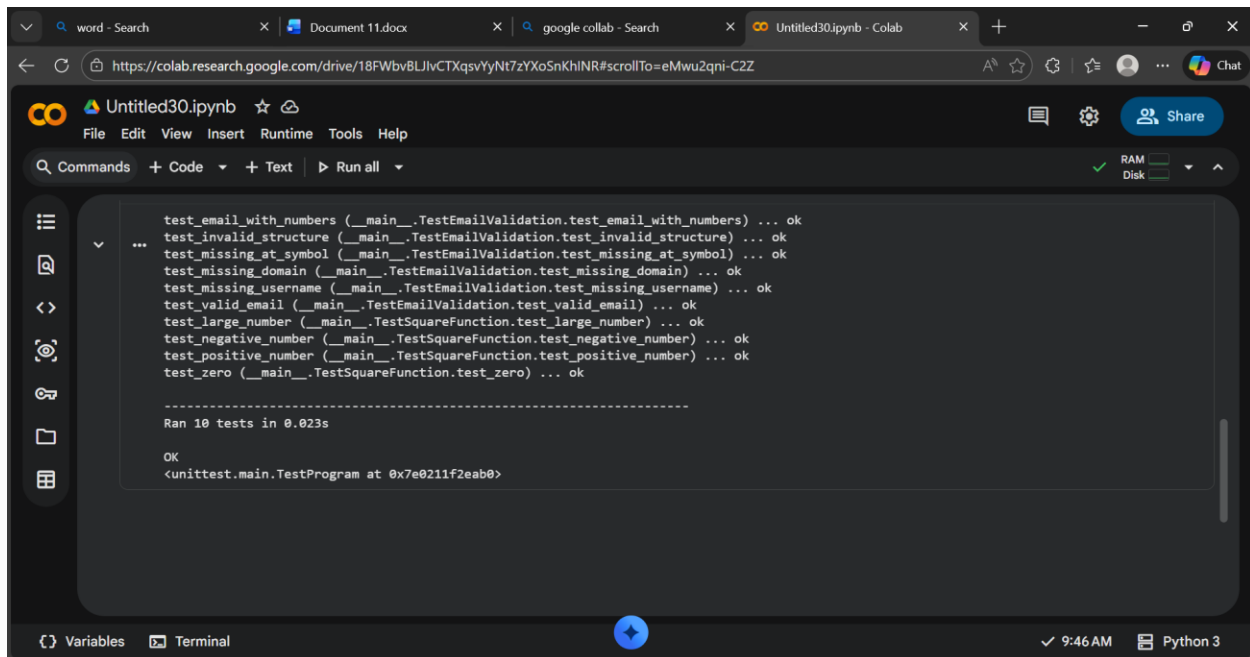
import re

# ----- 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

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

Output:



The screenshot shows a Google Colab notebook titled 'Untitled30.ipynb'. The code cell contains a series of unittest tests for email validation and a square function. The output shows that all 10 tests passed successfully, with a total execution time of 0.023 seconds. The tests include:

- test\_email\_with\_numbers
- test\_invalid\_structure
- test\_missing\_at\_symbol
- test\_missing\_domain
- test\_missing\_username
- test\_valid\_email
- test\_large\_number
- test\_negative\_number
- test\_positive\_number
- test\_zero

The output also shows the 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 image displays two sequential screenshots of a Google Colab notebook titled 'Untitled30.ipynb'. The interface includes a top toolbar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help' menus, and a bottom status bar showing '9:55 AM' and 'Python 3'.

**Top Screenshot:** 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 code cell [8] shows the implementation added after the tests:

```
#AI-Generated Implementation

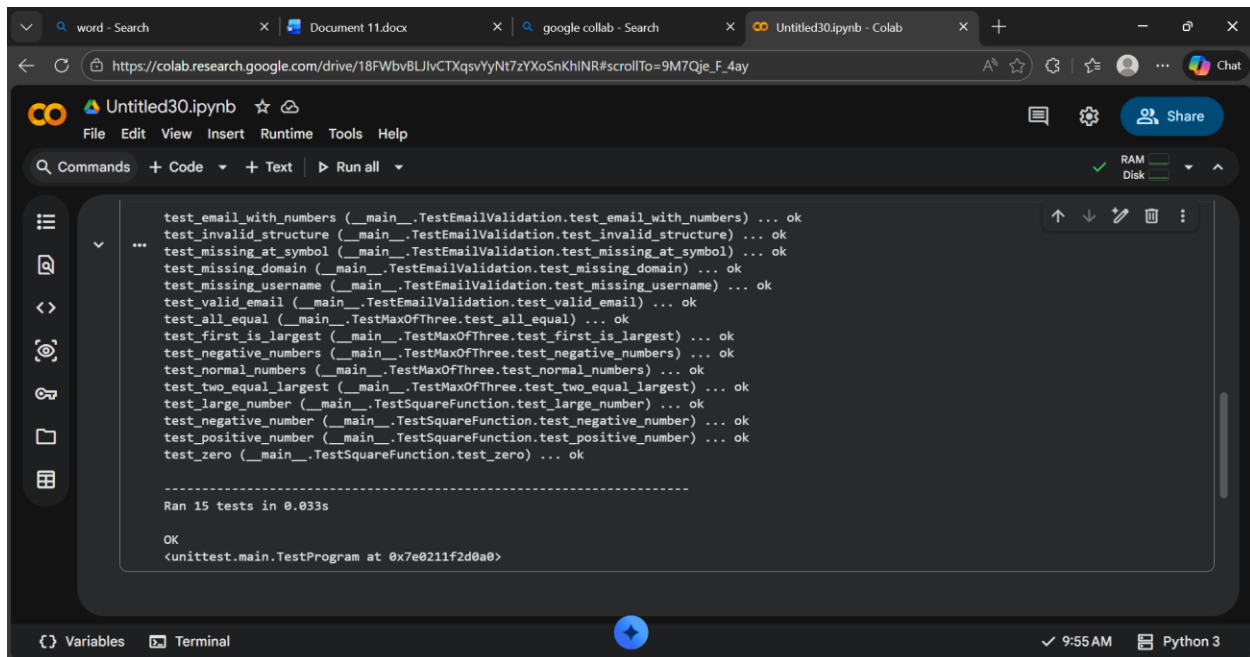
# ----- IMPLEMENTATION (AFTER TESTS) -----
def max_of_three(a, b, c):
    return max(a, b, c)
```

Below this, a new code cell [9] is added with the command to run the tests:

```
#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 a shopping cart module. The tests are organized into three groups: email validation, max of three, and square function. All tests pass, indicated by '... ok' at the end of each line. 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 class-based design.

**Top Screenshot:** The notebook is titled "Untitled30.ipynb". The code cell [10] shows the initial setup with imports and tests written first, following the TDD rule. The tests are:

```
[10] 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)
```

The status bar at the bottom indicates the runtime is at 9:58 AM and the environment is Python 3.

**Bottom Screenshot:** The notebook is still titled "Untitled30.ipynb". The code cell [11] shows the implementation added after the tests pass, following the TDD rule. The implementation is:

```
[11] # ----- 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 status bar at the bottom indicates the runtime is at 9:58 AM and the environment is Python 3.



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, titled '#Run Tests', contains `unittest.main(argv=[''], verbosity=2, exit=False)`. A 'Snipping Tool' notification is visible on the right, stating 'Screenshot copied to clipboard' and 'Automatically saved to screenshots folder.' The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all', and a sidebar with icons for file management and execution.

Output:

The screenshot shows the same Google Colab notebook, but now displaying the output of the test runner. The output lists 19 tests, all of which passed (e.g., `test_invalid_structure`, `test_missing_at_symbol`, `test_remove_item`). The summary line reads 'Ran 19 tests in 0.029s'. The bottom of the notebook shows the status bar with '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'. The interface includes a top toolbar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help' menus, and a bottom status bar showing 'Python 3' and the time '10:03 AM'.

**Top Screenshot:** The code cell [13] contains the following Python code:

```
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"))
```

**Bottom Screenshot:** The code cell [14] shows the implementation of the `is_palindrome` function:

```
#Ai Implemented Code

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

Below this, the code cell [15] runs the tests:

```
#Run Tests

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

Output:

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

https://colab.research.google.com/drive/18FWbvBLJlvCTXqsvYyNt7zYXoSnKhINR#scrollTo=LpQRy\_SmCH9E

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