AI ASSISTED CODING

LAB 8.4

NAME:K.Manikanta

ENROLL.NO:2403A5240

BATCH:03

TASK-01:

Write a test case to check if a function returns the square of a number.
Then write the function with help from GitHub Copilot or Cursor AI.

PROMPT:

Generate a python code to find the square of a given number like 2**2=4 ,3**2=9 and also give the test cases to check the efficiency of the code.

CODE:

```python
import unittest
import sys
import os
# Add the parent directory to the Python path to import square_function
sys.path.append(os.path.dirname(os.path.abspath(__file__)))
from square_function import square

class TestSquareFunction(unittest.TestCase):

    def test_positive_integer(self):
        """Tests squaring a positive integer."""
        self.assertEqual(square(5), 25)
        self.assertEqual(square(10), 100)

    def test_negative_integer(self):
        """Tests squaring a negative integer."""
        self.assertEqual(square(-4), 16)
        self.assertEqual(square(-1), 1)

    def test_zero(self):
        """Tests squaring zero."""
        self.assertEqual(square(0), 0)

    def test_float(self):
        """Tests squaring a floating-point number."""
        self.assertAlmostEqual(square(2.5), 6.25)
        self.assertAlmostEqual(square(-1.5), 2.25)

    def test_non_numeric_input_raises_error(self):
        """Tests that non-numeric input raises a TypeError."""
        # The 'with self.assertRaises(...)' block checks that the code
        # inside it raises the specified exception.
        with self.assertRaises(TypeError):
            square("a")
        with self.assertRaises(TypeError):
            square([2])

if __name__ == '__main__':
    unittest.main(verbosity=2)



def square(number):
    """
    Returns the square of a number.
    """
    return number * number
```

OUTPUT:

```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python313/python.exe c:/Users/ramch/OneDrive/Desktop/ai/test_square_function.
py
test_float (__main__.TestSquareFunction.test_float)
Tests squaring a floating-point number. ... ok
test_negative_integer (__main__.TestSquareFunction.test_negative_integer)
Tests squaring a negative integer. ... ok
test_non_numeric_input_raises_error (__main__.TestSquareFunction.test_non_numeric_input_raises_error)
Tests that non-numeric input raises a TypeError. ... ok
test_positive_integer (__main__.TestSquareFunction.test_positive_integer)
Tests squaring a positive integer. ... ok
test_zero (__main__.TestSquareFunction.test_zero)
Tests squaring zero. ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The code generated by the ai is efficient to calculate the square of the given number. And it also gave the test cases to find the efficiency of the code it also embedded the code with the particular print messages according to the datatype.

TASK-02:

Create test cases to validate an email address (e.g., contains @ and .com).Use AI assistance to implement the validate_email() function

PROMPT:

Generate the python code of test cases to test whether the email entered is correct or not. Use appropriate conditions mail like @ should present in it and .com should present in it.

CODE:

email_validator.py > ...

```python
import re

def validate_email(email):
    """
    Validates an email address based on a simple format check.

    Args:
        email: The string to validate.

    Returns:
        True if the email has a valid format, False otherwise.

    Raises:
        TypeError: If the input is not a string.
    """
    if not isinstance(email, str):
        raise TypeError("Input must be a string.")

    # A simple regex to check for a basic email structure: user@domain.tld
    # It checks for at least one character before @, after @, and after the dot.
    pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"
    return re.match(pattern, email) is not None
```

test_email_validator.py > TestEmailValidator > test_missing_domain

```python
import unittest
import sys
import os

# Add the parent directory to the Python path to import the function
sys.path.append(os.path.dirname(os.path.abspath(__file__)))

from email_validator import validate_email

class TestEmailValidator(unittest.TestCase):

    def test_valid_email(self):
        """Tests a standard, valid email address."""
        self.assertTrue(validate_email("test.user@example.com"))

    def test_missing_at_symbol(self):
        """Tests an email that is missing the '@' symbol."""
        self.assertFalse(validate_email("test.userexample.com"))

    def test_missing_domain_dot(self):
        """Tests an email that is missing the '.' in the domain part."""
        self.assertFalse(validate_email("test@examplecom"))

    def test_missing_username(self):
        """Tests an email that is missing the username part."""
        self.assertFalse(validate_email("@example.com"))

    def test_missing_domain(self):
        """Tests an email that is missing the domain name."""
        self.assertFalse(validate_email("test@.com"))

    def test_non_string_input(self):
        """Tests that non-string input raises a TypeError."""
        with self.assertRaises(TypeError):
            validate_email(12345)
        with self.assertRaises(TypeError):
            validate_email(None)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

OUTPUT:

```
OK
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python313/python.exe c:/Users/ramch/OneDrive/Desktop/ai/test_email_validator.py
test_missing_at_symbol (__main__.TestEmailValidator.test_missing_at_symbol)
Tests an email that is missing the '@' symbol. ... ok
test_missing_domain (__main__.TestEmailValidator.test_missing_domain)
Tests an email that is missing the domain name. ... ok
test_missing_domain_dot (__main__.TestEmailValidator.test_missing_domain_dot)
Tests an email that is missing the '.' in the domain part. ... ok
test_missing_username (__main__.TestEmailValidator.test_missing_username)
Tests an email that is missing the username part. ... ok
test_non_string_input (__main__.TestEmailValidator.test_non_string_input)
Tests that non-string input raises a TypeError. ... ok
test_valid_email (__main__.TestEmailValidator.test_valid_email)
Tests a standard, valid email address. ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.003s

OK
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The code generated by the ai provided many test cases to check whether the entered email is correct or not . It has used many conditions like @ should be present in the email entered.

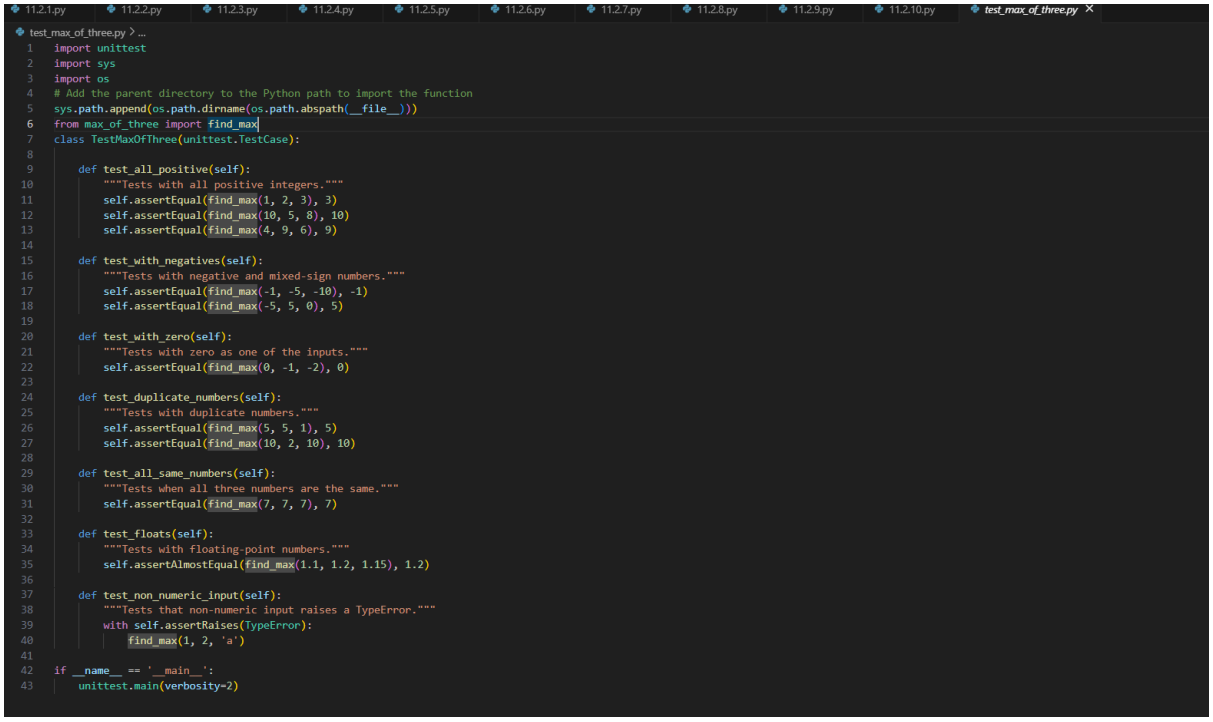
TASK-03:

Write test cases for a function that returns the maximum of three numbers. Prompt Copilot/Cursor to write the logic based on tests

PROMPT:

Generate the test cases to check whether the maximum number of the entered three numbers.

CODE:



```python
import unittest
import sys
import os
# Add the parent directory to the Python path to import the function
sys.path.append(os.path.dirname(os.path.abspath(__file__)))
from max_of_three import find_max
class TestMaxOfThree(unittest.TestCase):

    def test_all_positive(self):
        """Tests with all positive integers."""
        self.assertEqual(find_max(1, 2, 3), 3)
        self.assertEqual(find_max(10, 5, 8), 10)
        self.assertEqual(find_max(4, 9, 6), 9)

    def test_with_negatives(self):
        """Tests with negative and mixed-sign numbers."""
        self.assertEqual(find_max(-1, -5, -10), -1)
        self.assertEqual(find_max(-5, 5, 0), 5)

    def test_with_zero(self):
        """Tests with zero as one of the inputs."""
        self.assertEqual(find_max(0, -1, -2), 0)

    def test_duplicate_numbers(self):
        """Tests with duplicate numbers."""
        self.assertEqual(find_max(5, 5, 1), 5)
        self.assertEqual(find_max(10, 2, 10), 10)

    def test_all_same_numbers(self):
        """Tests when all three numbers are the same."""
        self.assertEqual(find_max(7, 7, 7), 7)

    def test_floats(self):
        """Tests with floating-point numbers."""
        self.assertAlmostEqual(find_max(1.1, 1.2, 1.15), 1.2)

    def test_non_numeric_input(self):
        """Tests that non-numeric input raises a TypeError."""
        with self.assertRaises(TypeError):
            find_max(1, 2, 'a')

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

```python
def find_max(a, b, c):
    """
    Finds the maximum of three numbers.

    Args:
        a: The first number.
        b: The second number.
        c: The third number.

    Returns:
        The largest of the three numbers.
    """
    # Python's built-in max() function is the most efficient way to do this.
    # It correctly handles integers, floats, and raises a TypeError for mixed types like number and string.
    return max(a, b, c)
```

OUTPUT:

```
OK
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python313/python.exe c:/Users/ramch/OneDrive/Desktop/ai/test_max_of_three.py
test_all_positive (__main__.TestMaxOfThree.test_all_positive)
Tests with all positive integers. ... ok
test_all_same_numbers (__main__.TestMaxOfThree.test_all_same_numbers)
Tests when all three numbers are the same. ... ok
test_duplicate_numbers (__main__.TestMaxOfThree.test_duplicate_numbers)
Tests with duplicate numbers. ... ok
test_floats (__main__.TestMaxOfThree.test_floats)
Tests with floating-point numbers. ... ok
test_non_numeric_input (__main__.TestMaxOfThree.test_non_numeric_input)
Tests that non-numeric input raises a TypeError. ... ok
test_with_negatives (__main__.TestMaxOfThree.test_with_negatives)
Tests with negative and mixed-sign numbers. ... ok
test_with_zero (__main__.TestMaxOfThree.test_with_zero)
Tests with zero as one of the inputs. ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The test cases generated by the ai are more efficient of check whether the maximum of the three numbers. It has given many more conditions to test the function.

TASK-04:

Use TDD to write a shopping cart class with methods to add, remove, and get total price. First write tests for each method, then generate code using AI

PROMPT:

Generate the python code for the shopping cart which add, remove, total price of the items which are present in the cart. Also give the test cases to check whether the given functions are in an efficient way or not.

CODE:

test_shopping_cart.py > ...

```python
import unittest
import sys
import os

# Add the parent directory to the Python path to import the class
sys.path.append(os.path.dirname(os.path.abspath(__file__)))

from shopping_cart import ShoppingCart

class TestShoppingCart(unittest.TestCase):

    def setUp(self):
        """Set up a new shopping cart for each test."""
        self.cart = ShoppingCart()

    def test_add_item(self):
        """Tests adding a new item to the cart."""
        self.cart.add_item("apple", 0.50)
        self.assertIn("apple", self.cart.items)
        self.assertEqual(self.cart.items["apple"]["price"], 0.50)
        self.assertEqual(self.cart.items["apple"]["quantity"], 1)

    def test_add_existing_item(self):
        """Tests adding an item that is already in the cart."""
        self.cart.add_item("banana", 0.30)
        self.cart.add_item("banana", 0.30)
        self.assertEqual(self.cart.items["banana"]["quantity"], 2)

    def test_remove_item(self):
        """Tests removing an item from the cart."""
        self.cart.add_item("orange", 0.75)
        self.cart.remove_item("orange")
        self.assertNotIn("orange", self.cart.items)

    def test_remove_one_of_multiple_items(self):
        """Tests decrementing the quantity when removing one of multiple items."""
        self.cart.add_item("apple", 0.50, 3) # Add 3 apples
        self.cart.remove_item("apple")
        self.assertEqual(self.cart.items["apple"]["quantity"], 2)

    def test_remove_nonexistent_item(self):
        """Tests that trying to remove an item not in the cart raises an error."""
        with self.assertRaises(KeyError):
            self.cart.remove_item("grape")

    def test_get_total_price_empty_cart(self):
        """Tests the total price of an empty cart."""
        self.assertEqual(self.cart.get_total_price(), 0)

    def test_get_total_price_with_items(self):
        """Tests the total price with multiple items and quantities."""
        self.cart.add_item("apple", 0.50, 2)  # 2 * 0.50 = 1.00
        self.cart.add_item("banana", 0.30, 3) # 3 * 0.30 = 0.90
        self.assertAlmostEqual(self.cart.get_total_price(), 1.90)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

0 ⚠ 0

shopping_cart.py > ShoppingCart

```python
class ShoppingCart:
    """
    A simple shopping cart class to manage items and calculate the total price.
    """
    def __init__(self):
        """Initializes an empty shopping cart."""
        self.items = {}

    def add_item(self, item_name, price, quantity=1):
        """
        Adds an item to the cart or updates its quantity if it already exists.

        Args:
            item_name (str): The name of the item.
            price (float): The price of a single item.
            quantity (int): The number of items to add.
        """
        if item_name in self.items:
            self.items[item_name]["quantity"] += quantity
        else:
            self.items[item_name] = {"price": price, "quantity": quantity}

    def remove_item(self, item_name, quantity=1):
        """
        Removes a given quantity of an item from the cart.
        If the quantity to remove is greater than or equal to the quantity
        in the cart, the item is removed completely.

        Args:
            item_name (str): The name of the item to remove.
            quantity (int): The number of items to remove.

        Raises:
            KeyError: If the item is not in the cart.
        """
        if item_name not in self.items:
            raise KeyError(f"'{item_name}' not found in cart.")

        if self.items[item_name]["quantity"] <= quantity:
            del self.items[item_name]
        else:
            self.items[item_name]["quantity"] -= quantity

    def get_total_price(self):
        """Calculates the total price of all items in the cart."""
        total = 0.0
        for item in self.items.values():
            total += item["price"] * item["quantity"]
        return total
```

OUTPUT:

```
OK
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python313/python.exe c:/Users/ramch/OneDrive/Desktop/ai/test_shopping_cart.py
test_add_existing_item (__main__.TestShoppingCart.test_add_existing_item)
Tests adding an item that is already in the cart. ... ok
test_add_item (__main__.TestShoppingCart.test_add_item)
Tests adding a new item to the cart. ... ok
test_get_total_price_empty_cart (__main__.TestShoppingCart.test_get_total_price_empty_cart)
Tests the total price of an empty cart. ... ok
test_get_total_price_with_items (__main__.TestShoppingCart.test_get_total_price_with_items)
Tests the total price with multiple items and quantities. ... ok
test_remove_item (__main__.TestShoppingCart.test_remove_item)
Tests removing an item from the cart. ... ok
test_remove_nonexistent_item (__main__.TestShoppingCart.test_remove_nonexistent_item)
Tests that trying to remove an item not in the cart raises an error. ... ok
test_remove_one_of_multiple_items (__main__.TestShoppingCart.test_remove_one_of_multiple_items)
Tests decrementing the quantity when removing one of multiple items. ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.002s

OK
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The code given by the ai is in an efficient way to find the total bills, adding items , remove items. And the test cases generated by it are more efficient to find the efficiency of the code.

TASK-05:

Write tests for a palindrome checker (e.g., is_palindrome("level") → True). Let Copilot suggest the function based on test case expectations.

PROMPT:

Write a python code to find whether the entered number or word is a palindrome or not also generate the test cases to find the efficiency of the code.

CODE:

test_palindrome_checker.py > TestPalindromeChecker > test_simple_palindrome

```python
import unittest
import sys
import os

# Add the parent directory to the Python path to import the function
sys.path.append(os.path.dirname(os.path.abspath(__file__)))

from palindrome_checker import is_palindrome

class TestPalindromeChecker(unittest.TestCase):

    def test_simple_palindrome(self):
        """Tests a simple, lowercase palindrome."""
        self.assertTrue(is_palindrome("level"))
        self.assertTrue(is_palindrome("madam"))

    def test_non_palindrome(self):
        """Tests a word that is not a palindrome."""
        self.assertFalse(is_palindrome("hello"))

    def test_case_insensitivity(self):
        """Tests that the check is case-insensitive."""
        self.assertTrue(is_palindrome("Racecar"))

    def test_with_spaces(self):
        """Tests a palindrome that includes spaces."""
        self.assertTrue(is_palindrome("nurses run"))

    def test_with_punctuation(self):
        """Tests a palindrome that includes punctuation and spaces."""
        self.assertTrue(is_palindrome("A man, a plan, a canal: Panama"))

    def test_empty_string(self):
        """Tests that an empty string is considered a palindrome."""
        self.assertTrue(is_palindrome(""))

    def test_non_string_input(self):
        """Tests that non-string input raises a TypeError."""
        with self.assertRaises(TypeError):
            is_palindrome(121)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

palindrome_checker.py > ...

```python
import re

def is_palindrome(s):
    """
    Checks if a string is a palindrome, ignoring case, spaces, and punctuation.

    Args:
        s (str): The string to check.

    Returns:
        bool: True if the string is a palindrome, False otherwise.
    """
    if not isinstance(s, str):
        raise TypeError("Input must be a string.")

    # 1. Normalize the string: remove non-alphanumeric characters and convert to lowercase.
    normalized_s = re.sub(r'[^a-zA-Z0-9]', '', s).lower()

    # 2. Check if the normalized string is equal to its reverse.
    return normalized_s == normalized_s[::-1]
```

OUTPUT:

OBSERVATION:

The ai generated code which checks whether the entered number or palindrome is palindrome or not. It generated a function which takes an input and checks for the palindrome and gives the appropriate result. It also generated the test cases to check the accuracy of the code.