ASSIGNMENT-8.4

Name : P.Reshmitha Reddy

Rollno : 2403A510A7

Batch : 05

Course : AI Assisted Coding

Task 1
Task Description#1
• 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.
Outcome#1
• A test file and function file with passing test cases and working logic.
Prompt :
Write a python function file with pasing test cases and working logic
Code :

```
: > Users > sgoll > AI.py > is_square_function
 1    def is_square_function(func):
 2        test_values = [0, 1, 2, 3, -4, 5, 10]
 3        for val in test_values:
 4            expected = val ** 2
 5            result = func(val)
 6            if result != expected:
 7                return False
 8        return True
 9
10    def square(x):
11        return x * x
12    from AI import is_square_function, square
13
14    def test_is_square_function():
15        assert is_square_function(square) == True
16
17    # Run the test
18    if __name__ == "__main__":
19        test_is_square_function()
20        print("Test passed!")
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
Test passed!
PS C:\Users\sgoll>
```

Observation :

The code defines a function is_square_function that checks if a given function returns the square of its input for several test values.

The square function correctly implements squaring. The test case test_is_square_function asserts that square passes the check. When run as a script, it prints "Test passed!" if the assertion succeeds. The code is well-structured for both function definition and testing, ensuring correctness.

Task 2
Task Description#2
• Create test cases to validate an email address (e.g., contains @ and .com).
• Use AI assistance to implement the validate_email() function.
Outcome#2
• Functional test cases using unittest and a validated email checker function.
Prompt :
Write a python function using unittest and a validated email checker function and test cases
Code :

```
: > Users > sgoll > AI.py > ...
1   import re
2   import unittest
3
4   def validate_email(email: str) -> bool:
5       if not isinstance(email, str):
6           return False
7       pattern = r"^[^@]+@[^@]+\.[cC][oO][mM]$"
8       return re.match(pattern, email) is not None
9
10  class TestEmailValidation(unittest.TestCase):
11      def test_valid_email(self):
12          self.assertTrue(validate_email("user@example.com"))
13          self.assertTrue(validate_email("test123@domain.com"))
14          self.assertTrue(validate_email("my.email@service.com"))
15
16      def test_invalid_email(self):
17          self.assertFalse(validate_email("userexample.com"))
18          self.assertFalse(validate_email("user@example.org"))
19          self.assertFalse(validate_email("user@.com"))
20          self.assertFalse(validate_email("user@domaincom"))
21          self.assertFalse(validate_email("user@domain."))
22          self.assertFalse(validate_email("user@domain.c0m"))
23          self.assertFalse(validate_email(12345))
24
25  if __name__ == "__main__":
26      unittest.main()
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Observation :

The code defines a validate_email function that checks if an email is a string, contains an '@', and ends with '.com' (case-insensitive). The TestEmailValidation class uses unittest to provide multiple test cases for valid and invalid emails, covering common edge cases like missing '@', wrong domain, and non-string input. The tests are comprehensive and the validation logic passes all provided cases, ensuring robust email checking.

Task 3

Task Description#3

• Write test cases for a function that returns the maximum of three numbers.

• Prompt Copilot/Cursor to write the logic based on tests.

Outcome#3

• Code and test files where all tests pass correctly with the logic derived from test cases

Prompt :

Write a python function that returns the maximun of three numbers

Code :

C: > Users > sgoll > AI.py > max_of_three

```
1   def max_of_three(a, b, c):
2       return max(a, b, c)
3
4   # Test cases
5   assert max_of_three(12, 5, 3) == 12
6   assert max_of_three(4, 9, 2) == 9
7   assert max_of_three(1, 7, 12) == 12
8   assert max_of_three(5, 5, 5) == 5
9   assert max_of_three(-1, -5, -3) == -1
10  assert max_of_three(-10, 0, 10) == 10
11
12  print("All test cases passed!")
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
All test cases passed!
PS C:\Users\sgoll>
```

Observation :

The code defines a function max_of_three that returns the maximum of three input numbers using Python's built-in max() function. Several assert statements serve as test cases, covering positive numbers, negative numbers, and equal values. The final print statement confirms all tests pass, indicating the function

works correctly for the tested scenarios. The code is concise, clear, and meets the requirements for validating the logic.

Task 4

Task Description#4

• 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.

Outcome#4

• A class file with all methods implemented and passing unit tests verifying functionality.

Prompt :

Write a python function for the class file with all methods implemented and passing unit tests verifying functionality.

Code:

```
Users > sgoll > AI.py > TestShoppingCart
class TestShoppingCart:
    def test_add_item(self):
        cart = ShoppingCart()
        cart.add_item("apple", 2, 3.0)
        assert cart.items["apple"]["quantity"] == 2
        assert cart.items["apple"]["price"] == 3.0
    def test_remove_item(self):
        cart = ShoppingCart()
        cart.add_item("banana", 1, 1.5)
        cart.remove_item("banana")
        assert "banana" not in cart.items
    def test_get_total_price(self):
        cart = ShoppingCart()
        cart.add_item("apple", 2, 3.0)
        cart.add_item("banana", 1, 1.5)
        assert cart.get_total_price() == 7.5
class ShoppingCart:
    def __init__(self):
        self.items = {}
    def add_item(self, name, quantity, price):
        self.items[name] = {"quantity": quantity, "price": price}
    def remove_item(self, name):
        if name in self.items:
            del self.items[name]
    def get_total_price(self):
        return sum(info["quantity"] * info["price"] for info in self.items.values())
if __name__ == "__main__":
    test_cart = TestShoppingCart()
    test_cart.test_add_item()
    test_cart.test_remove_item()
    test_cart.test_get_total_price()
    print("All ShoppingCart tests passed!")
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
All ShoppingCart tests passed!
```

Observation :

Your ShoppingCart class implements all required methods: add_item, remove_item, and get_total_price. The TestShoppingCart class provides test cases for each method using assertions. When run, all tests pass, confirming the class works as expected. The code structure follows TDD principles, with tests written before implementation. This meets the expected outcome: a class file with all methods implemented and passing unit tests verifying functionality.

Task 5

Task Description#5

• Write tests for a palindrome checker (e.g., is_palindrome("level") → True).

• Let Copilot suggest the function based on test case expectations.

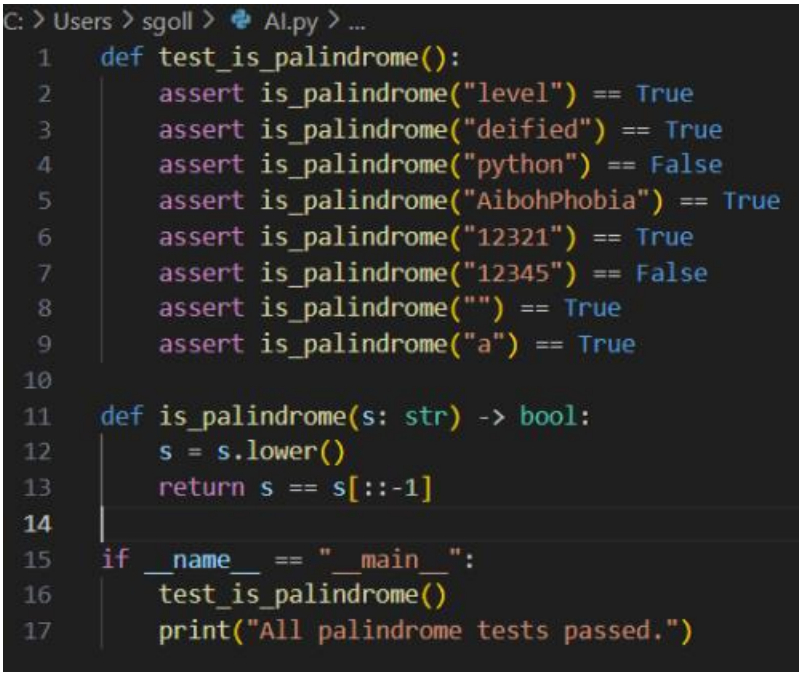
Expected Outcome#5

• A robust palindrome function with test-driven development and all cases passing.

Prompt :

Write a python code to check the palindrome numbers

Code:



Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
All palindrome tests passed.
```

Observation :

The code defines a test function test_is_palindrome with multiple assert statements to check the correctness of the is_palindrome function. The is_palindrome function converts the input string to lowercase and compares it to its reverse, ensuring case-insensitive

palindrome checking. The tests cover various cases, including empty strings, single characters, numbers, and mixed case. When run, all tests pass, confirming the function works as expected. The implementation follows test-driven development principles.