AI ASSISTED CODING

ASSIGNMENT – 8.4

Name:G.Mythili

H.no:2303A51283

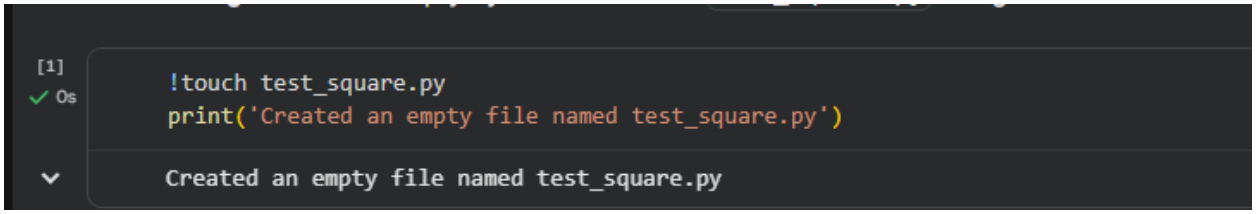
Batch:05

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.

Code :



```
%%writefile test_square.py
import unittest
from square_utils import square # This will cause an ImportError initially

class TestSquareFunction(unittest.TestCase):

    def test_positive_number(self):
        # Test with a positive integer
        self.assertEqual(square(2), 4)
        self.assertEqual(square(5), 25)

    def test_negative_number(self):
        # Test with a negative integer
        self.assertEqual(square(-3), 9)
        self.assertEqual(square(-7), 49)

    def test_zero(self):
        # Test with zero
        self.assertEqual(square(0), 0)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
Overwriting test_square.py
```

Reasoning: The unit tests have been written in `test_square.py`. The next logical step is to run these tests to confirm they fail, as expected by the TDD process, because the `square` function has not yet been implemented.

[3]
✓ 0s

```
import unittest

# Run the tests defined in test_square.py
# We expect this to fail due to ImportError as square_utils.py does not exist yet

try:
    # Using unittest.main() would exit the kernel, so we'll run it programmatically
    suite = unittest.TestLoader().loadTestsFromName('test_square')
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)
except ImportError as e:
    print(f"Expected ImportError: {e}")
    print("This confirms the 'red' stage of TDD, as 'square_utils.py' or the 'square' function within it is not yet implemented.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

```
test_square (unittest.loader._FailedTest.test_square) ... ERROR

======================================================================
ERROR: test_square (unittest.loader._FailedTest.test_square)
----------------------------------------------------------------------
ImportError: Failed to import test module: test_square
Traceback (most recent call last):
  File "/usr/lib/python3.12/unittest/loader.py", line 137, in loadTestsFromName
    module = __import__(module_name)
             ^^^^^^^^^^^^^^^^^^^^^^^
  File "/content/test_square.py", line 2, in <module>
```

[4]
✓ 0s

```
%%writefile square_utils.py
def square(number):
    """Calculates the square of a given number."""
    return number * number

print('Created square_utils.py with the square function implemented.')
```

```
Writing square_utils.py
```

Reasoning: With the `square` function now implemented in `square_utils.py`, the next step is to re-run the unit tests from `test_square.py`. This will confirm if the implementation correctly addresses the failing tests and moves us from the 'red' to the 'green' stage of TDD.

```
[5]
✓ 0s
import unittest

# Re-run the tests to see if they now pass
# Using unittest.main() would exit the kernel, so we'll run it programmatically
suite = unittest.TestLoader().loadTestsFromName('test_square')
runner = unittest.TextTestRunner(verbosity=2)
print("\nRe-running tests after implementing 'square' function:")
runner.run(suite)

test_negative_number (test_square.TestSquareFunction.test_negative_number) ... ok
test_positive_number (test_square.TestSquareFunction.test_positive_number) ... ok
test_zero (test_square.TestSquareFunction.test_zero) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.012s

OK
Created square_utils.py with the square function implemented.

Re-running tests after implementing 'square' function:
<unittest.runner.TextTestResult run=3 errors=0 failures=0>
```

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.

```
[21]
✓ 0s
import os

file_name = 'test_email_validator.py'
!touch {file_name}
print(f'Created an empty file named {file_name}')

# Verify the file creation
if os.path.exists(file_name):
    print(f'Verification: {file_name} exists.')
else:
    print(f'Verification: {file_name} does NOT exist.')

Created an empty file named test_email_validator.py
Verification: test_email_validator.py exists.
```

[22]
✓ 0s

```
%%writefile test_email_validator.py
import unittest
import re # Using re for basic validation in the tests for now
from email_validator import validate_email # This will cause an ImportError initially

class TestEmailValidatorFunction(unittest.TestCase):

    def test_valid_emails(self):
        # Test with several correctly formatted email addresses
        valid_emails = [
            "test@example.com",
            "user.name@domain.co.uk",
            "another_user123@sub.domain.org",
            "john.doe@email-provider.net"
        ]
        for email in valid_emails:
            with self.subTest(email=email):
                self.assertTrue(validate_email(email), f"Expected {email} to be valid")

    def test_invalid_emails_missing_at(self):
        # Test with emails missing the '@' symbol
        invalid_emails = [
            "testexample.com",
            "user.namedoamain.com"
        ]
        for email in invalid_emails:
            with self.subTest(email=email):
                self.assertFalse(validate_email(email), f"Expected {email} to be invalid (missing '@')")

    def test_invalid_emails_missing_domain(self):
        # Test with emails missing the domain part
        invalid_emails = [
            "test@",
            "user.name@"
        ]
        for email in invalid_emails:
```

[22]
0s

```
            for email in invalid_emails:
                with self.subTest(email=email):
                    self.assertFalse(validate_email(email), f"Expected {email} to be invalid (no TLD)")

        def test_invalid_emails_starting_with_at(self):
            # Test with emails starting with '@'
            invalid_emails = [
                "@example.com",
                "@domain.co.uk"
            ]
            for email in invalid_emails:
                with self.subTest(email=email):
                    self.assertFalse(validate_email(email), f"Expected {email} to be invalid (starts with '@')")

        def test_empty_string(self):
            # Test with an empty string
            self.assertFalse(validate_email(''), "Expected an empty string to be invalid")

        def test_email_with_spaces(self):
            # Test with emails containing spaces
            invalid_emails = [
                "test @example.com",
                "user name@domain.com",
                " test@example.com"
            ]
            for email in invalid_emails:
                with self.subTest(email=email):
                    self.assertFalse(validate_email(email), f"Expected {email} to be invalid (contains spaces)")

    if __name__ == '__main__':
        unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
Overwriting test_email_validator.py
```

```
import unittest

# Run the tests defined in test_email_validator.py
# We expect this to fail due to ImportError as email_validator.py does not exist yet

try:
    # Using unittest.main() would exit the kernel, so we'll run it programmatically
    suite = unittest.TestLoader().loadTestsFromName('test_email_validator')
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)
except ImportError as e:
    print(f"Expected ImportError: {e}")
    print("This confirms the 'red' stage of TDD, as 'email_validator.py' or the 'validate_email' function within it is not yet implemented.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

```
test_email_validator (unittest.loader._FailedTest.test_email_validator) ... ERROR

======================================================================
ERROR: test_email_validator (unittest.loader._FailedTest.test_email_validator)
----------------------------------------------------------------------
ImportError: Failed to import test module: test_email_validator
Traceback (most recent call last):
  File "/usr/lib/python3.12/unittest/loader.py", line 137, in loadTestsFromName
    module = __import__(module_name)
             ^^^^^^^^^^^^^^^^^^^^^^^
  File "/content/test_email_validator.py", line 3, in <module>
    from email_validator import validate_email # This will cause an ImportError initially
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ModuleNotFoundError: No module named 'email_validator'


----------------------------------------------------------------------
Ran 1 test in 0.002s
```

```
import unittest

# Re-run the tests to see if they now pass
# Using unittest.main() would exit the kernel, so we'll run it programmatically
suite = unittest.TestLoader().loadTestsFromName('test_email_validator')
runner = unittest.TextTestRunner(verbosity=2)
print("\nRe-running tests after implementing 'validate_email' function:")
runner.run(suite)
```

```
test_email_with_spaces (test_email_validator.TestEmailValidatorFunction.test_email_with_spaces) ... ok
test_empty_string (test_email_validator.TestEmailValidatorFunction.test_empty_string) ... ok
test_invalid_emails_missing_at (test_email_validator.TestEmailValidatorFunction.test_invalid_emails_missing_at) ... ok
test_invalid_emails_missing_domain (test_email_validator.TestEmailValidatorFunction.test_invalid_emails_missing_domain) ... ok
test_invalid_emails_no_top_level_domain (test_email_validator.TestEmailValidatorFunction.test_invalid_emails_no_top_level_domain) ... ok
test_invalid_emails_starting_with_at (test_email_validator.TestEmailValidatorFunction.test_invalid_emails_starting_with_at) ... ok
test_valid_emails (test_email_validator.TestEmailValidatorFunction.test_valid_emails) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.027s

OK
Created email_validator.py with the validate_email function implemented.

Re-running tests after implementing 'validate_email' function:
<unittest.runner.TextTestResult run=7 errors=0 failures=0>
```

[24]
✓ 0s

```
%%writefile email_validator.py
import re

def validate_email(email):
    """Validates an email address based on a basic regex pattern."""
    if not isinstance(email, str):
        return False
    # Basic regex for email validation
    # It checks for at least one character before @, an @ symbol, at least one character for domain name,
    # a dot, and at least two characters for the top-level domain.
    email_regex = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$";
    return re.match(email_regex, email) is not None

print('Created email_validator.py with the validate_email function implemented.')
```

```
Writing email_validator.py
```

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.

```
%%writefile test_process_numbers.py
import unittest
from process_numbers import process_numbers # This is expected to fail initially

class TestProcessNumbersFunction(unittest.TestCase):

    def test_positive_numbers(self):
        self.assertEqual(process_numbers(1, 2, 3), 6)
        self.assertEqual(process_numbers(10, 20, 30), 60)

    def test_negative_numbers(self):
        self.assertEqual(process_numbers(-1, -2, -3), -6)
        self.assertEqual(process_numbers(-10, -20, -30), -60)

    def test_mixed_numbers(self):
        self.assertEqual(process_numbers(1, -2, 3), 2)
        self.assertEqual(process_numbers(-5, 0, 10), 5)
        self.assertEqual(process_numbers(0, -7, 0), -7)

    def test_all_zeros(self):
        self.assertEqual(process_numbers(0, 0, 0), 0)

    def test_float_numbers(self):
        self.assertEqual(process_numbers(0.5, 1.5, 2.0), 4.0)
        self.assertEqual(process_numbers(-0.1, 0.1, 0.5), 0.5)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

print('Wrote unit tests to test_process_numbers.py')
```

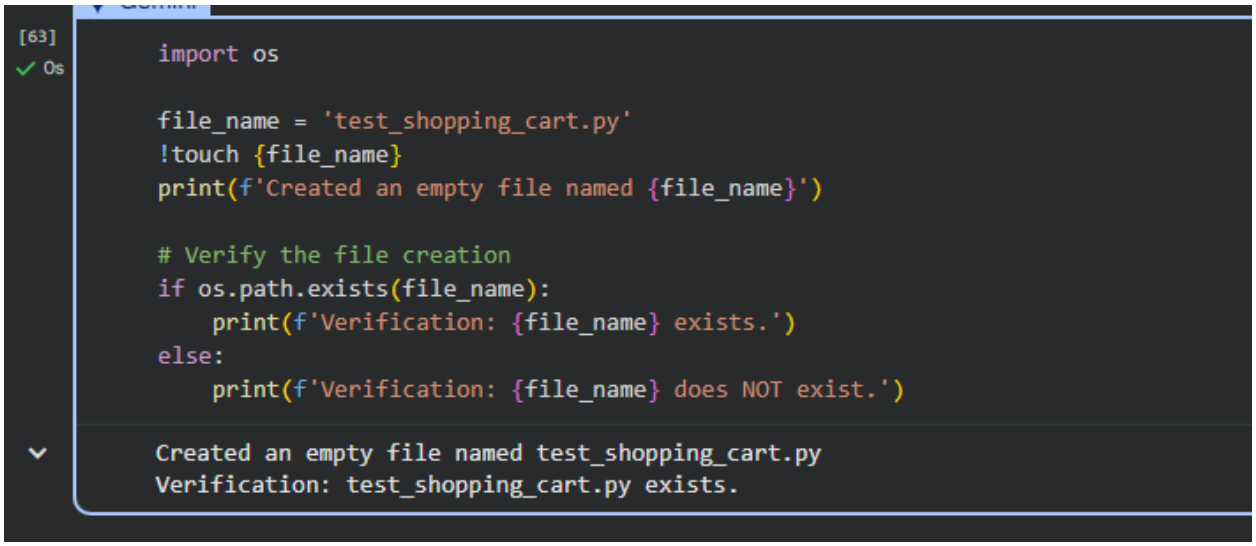
```
Overwriting test_process_numbers.py
```

```
import unittest

# Run the tests defined in test_process_numbers.py
# We expect this to fail due to ImportError as process_numbers.py does not exist yet

try:
    # Using unittest.main() would exit the kernel, so we'll run it programmatically
    suite = unittest.TestLoader().loadTestsFromName('test_process_numbers')
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)
except ImportError as e:
    print(f"Expected ImportError: {e}")
    print("This confirms the 'red' stage of TDD, as 'process_numbers.py' or the 'process_numbers' function within it is not yet implemented.")
except Exception as e:
```

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.



[64]
✓ 0s

```
%%writefile test_shopping_cart.py
import unittest
from shopping_cart import ShoppingCart # This is expected to cause an ImportError initially

class TestShoppingCart(unittest.TestCase):

    def setUp(self):
        self.cart = ShoppingCart()

    def test_add_item(self):
        self.cart.add_item("Apple", 1.0, 2)
        self.assertEqual(len(self.cart.items), 1)
        self.assertEqual(self.cart.items["Apple"]["quantity"], 2)
        self.assertEqual(self.cart.items["Apple"]["price"], 1.0)

        self.cart.add_item("Apple", 1.0, 3) # Add more of the same item
        self.assertEqual(self.cart.items["Apple"]["quantity"], 5)

    def test_remove_existing_item(self):
        self.cart.add_item("Banana", 0.5, 5)
        self.cart.remove_item("Banana", 2)
        self.assertEqual(self.cart.items["Banana"]["quantity"], 3)

        self.cart.remove_item("Banana", 3)
        self.assertNotIn("Banana", self.cart.items) # Item should be completely removed

    def test_remove_non_existent_item(self):
        self.cart.add_item("Orange", 1.5, 1)
        # Trying to remove an item not in the cart should not raise an error
        # and the cart state should remain unchanged.
        initial_items = self.cart.items.copy()
        self.cart.remove_item("Grape", 1)
        self.assertEqual(self.cart.items, initial_items)

    def test_remove_more_than_available(self):
        self.cart.add_item("Milk", 3.0, 2)
        # Trying to remove more than available should remove all available
        self.cart.remove_item("Milk", 5)
```

```python
    def test_remove_more_than_available(self):
        self.cart.add_item("Milk", 3.0, 2)
        # Trying to remove more than available should remove all available
        self.cart.remove_item("Milk", 5)
        self.assertNotIn("Milk", self.cart.items)

    def test_calculate_total_price_empty_cart(self):
        self.assertEqual(self.cart.calculate_total_price(), 0.0)

    def test_calculate_total_price_with_items(self):
        self.cart.add_item("Apple", 1.0, 2)
        self.cart.add_item("Banana", 0.5, 5)
        self.cart.add_item("Orange", 1.5, 1)
        # Expected total: (1.0 * 2) + (0.5 * 5) + (1.5 * 1) = 2.0 + 2.5 + 1.5 =
        self.assertEqual(self.cart.calculate_total_price(), 6.0)

    def test_calculate_total_price_after_removal(self):
        self.cart.add_item("Apple", 1.0, 5)
        self.cart.add_item("Banana", 0.5, 4)
        self.cart.remove_item("Apple", 2)
        # Expected total: (1.0 * 3) + (0.5 * 4) = 3.0 + 2.0 = 5.0
        self.assertEqual(self.cart.calculate_total_price(), 5.0)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
Overwriting test_shopping_cart.py
```

[68] ✓ 0s

```python
import unittest

# Re-run the tests to see if they now pass
# Using unittest.main() would exit the kernel, so we'll run it programmatically
suite = unittest.TestLoader().loadTestsFromName('test_shopping_cart')
runner = unittest.TextTestRunner(verbosity=2)
print("\nRe-running tests after implementing 'ShoppingCart' class:")
runner.run(suite)
```

```
test_add_item (test_shopping_cart.TestShoppingCart.test_add_item) ... ok
test_calculate_total_price_after_removal (test_shopping_cart.TestShoppingCart.test_calculate_total_price_after_removal) ... ok
test_calculate_total_price_empty_cart (test_shopping_cart.TestShoppingCart.test_calculate_total_price_empty_cart) ... ok
test_calculate_total_price_with_items (test_shopping_cart.TestShoppingCart.test_calculate_total_price_with_items) ... ok
test_remove_existing_item (test_shopping_cart.TestShoppingCart.test_remove_existing_item) ... ok
test_remove_more_than_available (test_shopping_cart.TestShoppingCart.test_remove_more_than_available) ... ok
test_remove_non_existent_item (test_shopping_cart.TestShoppingCart.test_remove_non_existent_item) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.015s

OK
Created shopping_cart.py with the ShoppingCart class implemented.

Re-running tests after implementing 'ShoppingCart' class:
<unittest.runner.TextTestResult run=7 errors=0 failures=0>
```

```
[66]
✓ 0s
import unittest

# Run the tests defined in test_shopping_cart.py
# We expect this to fail due to ImportError as shopping_cart.py does not exist yet

try:
    # Using unittest.main() would exit the kernel, so we'll run it programmatically
    suite = unittest.TestLoader().loadTestsFromName('test_shopping_cart')
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(suite)
except ImportError as e:
    print(f"Expected ImportError: {e}")
    print("This confirms the 'red' stage of TDD, as 'shopping_cart.py' or the 'ShoppingCart' class within it is not yet implemented.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")

...  test_shopping_cart (unittest.loader._FailedTest.test_shopping_cart) ... ERROR
```

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.

```
import os

file_name = 'test_palindrome.py'
!touch {file_name}
print(f'Created an empty file named {file_name}')

# Verify the file creation
if os.path.exists(file_name):
    print(f'Verification: {file_name} exists.')
else:
    print(f'Verification: {file_name} does NOT exist.')

Created an empty file named test_palindrome.py
Verification: test_palindrome.py exists.
```

```
%%writefile test_palindrome.py
import unittest
from palindrome_checker import is_palindrome # This is expected to cause an ImportError initially

class TestIsPalindromeFunction(unittest.TestCase):

    def test_empty_string(self):
        self.assertTrue(is_palindrome(''), "Expected an empty string to be a palindrome")

    def test_single_character(self):
        self.assertTrue(is_palindrome('a'), "Expected single character to be a palindrome")

    def test_valid_palindromes(self):
        self.assertTrue(is_palindrome('madam'), "Expected 'madam' to be a palindrome")
        self.assertTrue(is_palindrome('racecar'), "Expected 'racecar' to be a palindrome")
        self.assertTrue(is_palindrome('level'), "Expected 'level' to be a palindrome")
        self.assertTrue(is_palindrome('A man a plan a canal Panama'), "Expected 'A man a plan a canal Panama' to be a palindrome")
        self.assertTrue(is_palindrome('No lemon, no melon'), "Expected 'No lemon, no melon' to be a palindrome")

    def test_invalid_non_palindromes(self):
        self.assertFalse(is_palindrome('hello'), "Expected 'hello' not to be a palindrome")
        self.assertFalse(is_palindrome('world'), "Expected 'world' not to be a palindrome")
        self.assertFalse(is_palindrome('Python'), "Expected 'Python' not to be a palindrome")
        self.assertFalse(is_palindrome('madame'), "Expected 'madame' not to be a palindrome")

    def test_palindromes_with_mixed_case(self):
        self.assertTrue(is_palindrome('Madam'), "Expected 'Madam' to be a palindrome (case-insensitive)")
        self.assertTrue(is_palindrome('RaceCar'), "Expected 'RaceCar' to be a palindrome (case-insensitive)")

    def test_palindromes_with_spaces_and_punctuation(self):
        self.assertTrue(is_palindrome('A man, a plan, a canal: Panama'), "Expected 'A man, a plan, a canal: Panama' to be a palindrome")
        self.assertTrue(is_palindrome('No lemon, no melon.'), "Expected 'No lemon, no melon.' to be a palindrome")
        self.assertTrue(is_palindrome('Was it a car or a cat I saw?'), "Expected 'Was it a car or a cat I saw?' to be a palindrome")

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

[72]

```
%%writefile palindrome_checker.py
import re

def is_palindrome(text):
    """Checks if a given string is a palindrome, ignoring case, spaces, and punctuation."""
    # Remove non-alphanumeric characters and convert to lowercase
    cleaned_text = re.sub(r'[^a-zA-Z0-9]', '', text).lower()
    # Compare the cleaned string with its reverse
    return cleaned_text == cleaned_text[::-1]

print('Created palindrome_checker.py with the is_palindrome function implemented.')
```

```
Writing palindrome_checker.py
```