

AI Assisted Coding

Assignment 8.3

Name: V.Harivamsh

Hall ticket no: 2303A51266

Batch no: 19

Task 1:

Prompt:

Generate Python unittest test cases for validating email formats based on the following rules:

- Must contain @ and .
- Must not start or end with special characters
- Must not contain multiple @ symbols

Code & Output:

The screenshot shows a code editor with the file `assignment_8.3.py` open. The code implements a function `is_valid_email` and a class `TestEmailValidation` with various test methods. The terminal window below shows the command `python3.12.exe "C:/Users/hariv/OneDrive/Documents/SRU/3 year II sem/AI_Assistant_coding/assignment_8.3.py"` being run, and the output indicates 6 tests ran in 0.001s, all of which were OK.

```
# Task 1: Email Validation using TDD
import unittest

def is_valid_email(email):
    # Must contain exactly one '@'
    if email.count('@') != 1:
        return False
    # Must not start or end with special characters
    if email.startswith('(@', '.', '_')) or email.endswith('(@', '.', '_')):
        return False
    # Must contain '.' after '@'
    local_part, domain_part = email.split('@')
    if '.' not in local_part:
        return False
    return True

class TestEmailValidation(unittest.TestCase):
    def test_valid_email(self):
        self.assertTrue(is_valid_email("test@example.com"))
    def test_missing_at_symbol(self):
        self.assertFalse(is_valid_email("testexample.com"))
    def test_missing_dot(self):
        self.assertFalse(is_valid_email("test@examplecom"))
    def test_multiple_at_symbols(self):
        self.assertFalse(is_valid_email("test@example.com"))
    def test_starting_with_special_character(self):
        self.assertFalse(is_valid_email("@testexample.com"))
    def test_ending_with_special_character(self):
        self.assertFalse(is_valid_email("testexample.com"))

if __name__ == "__main__":
    unittest.main()
```

PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding> & C:/Users/hariv/AppData/Local/Microsoft/WindowsApps/python3.12.exe "C:/Users/hariv/OneDrive/Documents/SRU/3 year II sem/AI_Assistant_coding/assignment_8.3.py"
Ran 6 tests in 0.001s
OK
PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding>

Explanation:

In this task, the AI began by creating test cases that included both valid and invalid email formats. The implementation was then developed to meet all the defined test conditions.

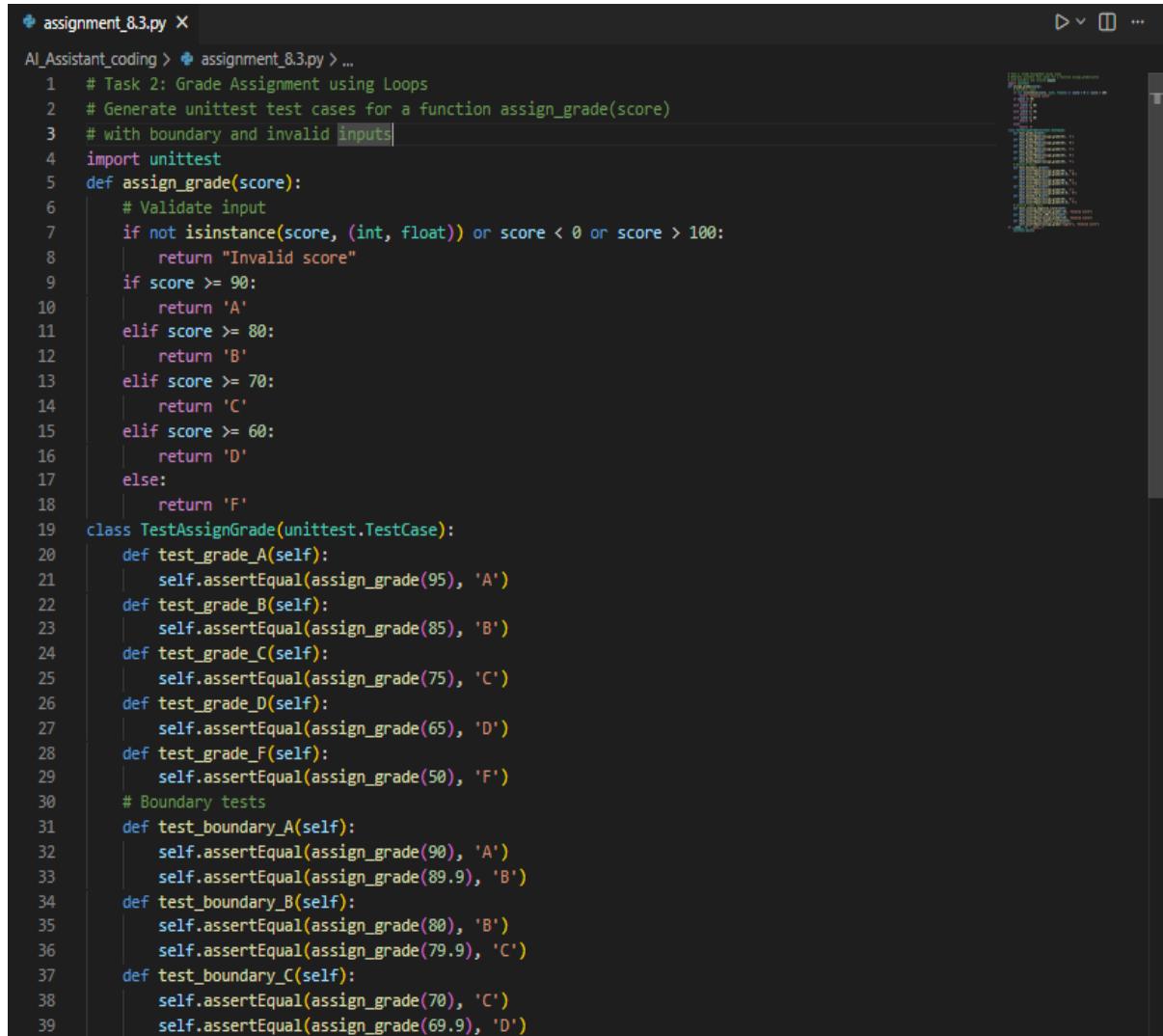
The function verifies the presence of "@" and ".", ensures that there is only one "@" symbol, and checks that the email does not begin or end with special characters. By following the Test-Driven Development (TDD) approach, the correctness of the solution was confirmed through successful test results.

Task 2: Grade Assignment using Loops

Prompt:

Generate unit test test cases for a function assign_grade(score) with boundary and invalid inputs.

Code & Output:



The screenshot shows a code editor window with a dark theme. The file is named 'assignment_83.py'. The code defines a function 'assign_grade(score)' which validates the input and returns a grade based on the score. It includes a class 'TestAssignGrade' with methods for testing various grades and boundary conditions. The right side of the screen shows a terminal window displaying the output of running the tests, showing green checkmarks for successful tests.

```
assignment_83.py X
AI-Assistant_coding > assignment_83.py > ...
1 # Task 2: Grade Assignment using Loops
2 # Generate unittest test cases for a function assign_grade(score)
3 # with boundary and invalid inputs
4 import unittest
5 def assign_grade(score):
6     # Validate input
7     if not isinstance(score, (int, float)) or score < 0 or score > 100:
8         return "Invalid score"
9     if score >= 90:
10        return 'A'
11    elif score >= 80:
12        return 'B'
13    elif score >= 70:
14        return 'C'
15    elif score >= 60:
16        return 'D'
17    else:
18        return 'F'
19 class TestAssignGrade(unittest.TestCase):
20     def test_grade_A(self):
21         self.assertEqual(assign_grade(95), 'A')
22     def test_grade_B(self):
23         self.assertEqual(assign_grade(85), 'B')
24     def test_grade_C(self):
25         self.assertEqual(assign_grade(75), 'C')
26     def test_grade_D(self):
27         self.assertEqual(assign_grade(65), 'D')
28     def test_grade_F(self):
29         self.assertEqual(assign_grade(50), 'F')
30     # Boundary tests
31     def test_boundary_A(self):
32         self.assertEqual(assign_grade(90), 'A')
33         self.assertEqual(assign_grade(89.9), 'B')
34     def test_boundary_B(self):
35         self.assertEqual(assign_grade(80), 'B')
36         self.assertEqual(assign_grade(79.9), 'C')
37     def test_boundary_C(self):
38         self.assertEqual(assign_grade(70), 'C')
39         self.assertEqual(assign_grade(69.9), 'D')
```

The screenshot shows a code editor with Python test code and a terminal window.

```
40     def test_boundary_D(self):
41         self.assertEqual(assign_grade(60), 'D')
42         self.assertEqual(assign_grade(59.9), 'F')
43     # Invalid input tests
44     def test_invalid_negative_score(self):
45         self.assertEqual(assign_grade(-10), "Invalid score")
46     def test_invalid_over_100_score(self):
47         self.assertEqual(assign_grade(110), "Invalid score")
48     def test_invalid_non_numeric_score(self):
49         self.assertEqual(assign_grade("eighty"), "Invalid score")
50 if __name__ == "__main__":
51     unittest.main()
```

TERMINAL

```
PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding> & C:/Users/hariv/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/hariv/OneDrive/Documents/SRU/3 year II sem/AI_Assistant_coding/assignment_8.3.py"
-----
Ran 12 tests in 0.001s
OK
PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding>
```

Explanation:

The AI-generated test cases covered normal score ranges, boundary conditions, and invalid inputs to ensure thorough evaluation. The implementation assigns grades accurately using conditional logic and also checks the input type and valid range to prevent incorrect or misleading results. Since all the test cases pass successfully, the solution's correctness is well confirmed.

Task 3: Sentence Palindrome Checker

Prompt:

Generate unittest test cases for checking whether a sentence is a palindrome while ignoring case, spaces, and punctuation.

Code & Output:

```

assignment_8.3.py
AI_Assistant_coding > assignment_8.3.py > ...
1 # Task 3: Sentence Palindrome Checker
2 # Generate unittest test cases for checking whether a sentence is a palindrome
3 # while ignoring case, spaces, and punctuation.
4 import unittest
5 import string
6 def is_palindrome(sentence):
7     # Remove non-alphanumeric characters and convert to lowercase
8     cleaned_sentence = ''.join(
9         char.lower() for char in sentence if char.isalnum()
10    )
11    return cleaned_sentence == cleaned_sentence[::-1]
12 class TestPalindromeChecker(unittest.TestCase):
13     def test_palindrome_sentence(self):
14         self.assertTrue(is_palindrome("A man, a plan, a canal, Panama"))
15     def test_non_palindrome_sentence(self):
16         self.assertFalse(is_palindrome("this is not a palindrome"))
17     def test_empty_string(self):
18         self.assertTrue(is_palindrome(""))
19     def test_single_character(self):
20         self.assertTrue(is_palindrome("x"))
21     def test_palindrome_with_numbers(self):
22         self.assertTrue(is_palindrome("12321"))
23     def test_non_palindrome_with_numbers(self):
24         self.assertFalse(is_palindrome("12345"))
25     def test_palindrome_with_mixed_characters(self):
26         self.assertTrue(is_palindrome("No 'x' in Nixon"))
27     def test_non_palindrome_with_mixed_characters(self):
28         self.assertFalse(is_palindrome("Hello, World!"))
29 if __name__ == "__main__":
30     unittest.main()

```

TERMINAL

```

PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding> & C:/Users/hariv/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/hariv/OneDrive/Documents/SRU/3 year II sem/AI_Assistant_coding/assignment_8.3.py"
* .....
-----
Ran 8 tests in 0.002s
OK
PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding>

```

Explanation:

The AI-generated test cases cover both palindromic and non-palindromic sentences to ensure proper validation. In the implementation, spaces and punctuation are removed using regular expressions, and the text is converted to lowercase for consistency. The processed string is then compared with its reversed version. This approach guarantees accurate palindrome detection, regardless of the original formatting.

Task 4: ShoppingCart Class

Prompt:

Generate unittest test cases for a ShoppingCart class with add_item, remove_item, and total_cost methods.

Code & Output:

```

❸ assignment_8.3.py ×
AI_Assistant_coding > ❹ assignment_8.3.py > ...
1 # Generate unittest test cases for a ShoppingCart class
2 # with add_item, remove_item, and total_cost methods.
3 import unittest
4
5 class ShoppingCart:
6     def __init__(self):
7         self.items = {}
8
9     def add_item(self, item_name, price):
10        if item_name in self.items:
11            self.items[item_name] += price
12        else:
13            self.items[item_name] = price
14
15    def remove_item(self, item_name):
16        if item_name in self.items:
17            del self.items[item_name]
18
19    def total_cost(self):
20        return sum(self.items.values())
21
22 class TestShoppingCart(unittest.TestCase):
23     def setUp(self):
24         self.cart = ShoppingCart()
25
26     def test_add_item(self):
27         self.cart.add_item("Apple", 1.00)
28         self.assertEqual(self.cart.items, {"Apple": 1.00})
29
30     def test_add_multiple_items(self):
31         self.cart.add_item("Apple", 1.00)
32         self.cart.add_item("Banana", 0.50)
33         self.assertEqual(self.cart.items, {"Apple": 1.00, "Banana": 0.50})
34
35     def test_remove_item(self):
36         self.cart.add_item("Apple", 1.00)
37         self.cart.remove_item("Apple")
38         self.assertEqual(self.cart.items, {})
39
40     def test_remove_nonexistent_item(self):
41         self.cart.add_item("Apple", 1.00)
42         self.cart.remove_item("Banana") # Should not crash
43         self.assertEqual(self.cart.items, {"Apple": 1.00})
44
45     def test_total_cost(self):
46         self.cart.add_item("Apple", 1.00)
47         self.cart.add_item("Banana", 0.50)
48         self.assertEqual(self.cart.total_cost(), 1.50)
49
50     def test_total_cost_empty_cart(self):
51         self.assertEqual(self.cart.total_cost(), 0)
52
53 if __name__ == "__main__":
54     unittest.main()

```

TERMINAL

```

PS C:\Users\hari\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding> & C:/Users/hari/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/hari/OneDrive/Documents\SRU\3 year II sem\AI_Assistant_coding\assignment_8.3.py"
• .....
----- Ran 6 tests in 0.001s
OK
PS C:\Users\hari\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding>

```

Explanation:

The AI-generated test cases thoroughly check item addition, item removal, and the calculation of the total cost. The class stores items and their prices in a dictionary, ensuring organized data management. Its methods properly update the cart's contents, and the total amount is calculated by summing all the stored values. Since the implementation successfully passes all the generated tests, its reliability is well established.

Task 5: Date Format Conversion

Prompt:

Generate unittest test cases for converting date format from YYYY-MM-DD to DD-MM-YYYY.

Code & Output:

The screenshot shows a code editor window with the file `assignment_8.3.py` open. The code implements a date format conversion function and includes several unit test cases using the `unittest` module. The terminal below shows the execution of the script and its successful completion.

```
❶ assignment_8.3.py ✘
AI_Assistant_coding > ❷ assignment_8.3.py > ...
1  # Task 5: Date Format Conversion
2  # Generate unittest test cases for converting date format from YYYY-MM-DD to DD-MM-YYYY.
3  import unittest
4  def convert_date_format(date_str):
5      if not isinstance(date_str, str):
6          return "Invalid date format"
7      parts = date_str.split('-')
8      if len(parts) != 3:
9          return "Invalid date format"
10     year, month, day = parts
11     if not (year.isdigit() and month.isdigit() and day.isdigit()):
12         return "Invalid date format"
13     if len(year) != 4 or len(month) != 2 or len(day) != 2:
14         return "Invalid date format"
15     return f'{day}-{month}-{year}'
16 class TestDateFormatConversion(unittest.TestCase):
17     def test_valid_date(self):
18         self.assertEqual(convert_date_format("2024-06-15"), "15-06-2024")
19     def test_invalid_date_format(self):
20         self.assertEqual(convert_date_format("15-06-2024"), "Invalid date format")
21     def test_non_string_input(self):
22         self.assertEqual(convert_date_format(20240615), "Invalid date format")
23     def test_empty_string(self):
24         self.assertEqual(convert_date_format(""), "Invalid date format")
25     def test_extra_characters(self):
26         self.assertEqual(convert_date_format("2024-06-15-01"), "Invalid date format")
27 if __name__ == "__main__":
28     unittest.main()

PROBLEMS OUTPUT TERMINAL PORTS ... | X
> TERMINAL
❸ PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding> & C:/Users/hariv/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/hariv/OneDrive/Documents\SRU\3 year II sem\AI_Assistant_coding\assignment_8.3.py"
...
Ran 5 tests in 0.001s
OK
❹ PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding>
```

Explanation:

The AI-generated test cases check both correctly formatted and invalid date inputs to ensure proper validation. The implementation separates the input string into parts and rearranges them into the desired format. It also includes error handling to return a clear message when the input format is incorrect. Since all the test cases pass successfully, the solution works as expected.

Final Conclusion:

This lab highlights the effectiveness of applying Test-Driven Development (TDD) with the support of AI. By creating test cases before writing the actual implementation, developers can better ensure correctness, reliability, and proper validation. While AI helps speed up the process of generating tests, human review is still crucial to guarantee comprehensive, meaningful, and well-designed test coverage.