

Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases

Name: Syed Murtaza

Hall ticket no: 2303A51259

Batch no: 19

Task 1: Email Validation using TDD

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 two panes. The left pane displays a Python file named `Assignment_8.3.py` containing test cases for email validation. The right pane shows the terminal output of running the tests.

```
Assignment_8.3.py
Assignment_8.3.py 100% 100% 100%
1 Assignment_8.3.py 2 Task 1: Email Validation using TDD
2
3 Generate Python unittest test cases for validating email formats based on the
4 following rules:
5 - Must contain @ and .
6 - Must not start or end with special characters
7 - Must not contain multiple @ symbols
8
9 import unittest
10 def is_valid_email(email):
11     if email.count('@') != 1:
12         return False
13     if email.startswith('@') or email.endswith('@') or email.startswith('.') or email.
14         endswith('.'):
15             return False
16     if '@' not in email.split('@')[1]:
17         return False
18     return True
19
20 class TestEmailValidation(unittest.TestCase):
21     def test_valid_email(self):
22         self.assertTrue(is_valid_email("test@example.com"))
23     def test_missing_at_symbol(self):
24         self.assertFalse(is_valid_email("testexample.com"))
25     def test_missing_dot(self):
26         self.assertFalse(is_valid_email("test@examplecom"))
27     def test_multiple_at_symbols(self):
28         self.assertFalse(is_valid_email("test@example.com"))
29     def test_starting_with_special_character(self):
30         self.assertFalse(is_valid_email("@testexample.com"))
31     def test_ending_with_special_character(self):
32         self.assertFalse(is_valid_email("test@example.com."))
33
34 if __name__ == '__main__':
35     unittest.main()

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3> python -u "e:\3rd Year\2nd Se
m\AI Assisted Coding\Assignment_8.3\Assignment_8.3.py"
.....
Ran 6 tests in 0.000s
OK
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3>
```

Explanation:

In this task, AI first generated test cases covering both valid and invalid email formats. The implementation was then written to satisfy all test conditions. The function checks for the presence of "@" and ".", prevents multiple "@" symbols, and ensures the email does not start or end with special characters. By following the TDD approach, correctness was validated through successful test execution.

Task 2: Grade Assignment using Loops

Prompt:

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

Code & Output:

```
Assignment_8.3.py
# Task 2: Grade Assignment using Loops
# Generate unittest test cases for a function assign_grade(score) with boundary and invalid inputs.

import unittest

def assign_grade(score):
    if not isinstance(score, (int, float)) or score < 0 or score > 100:
        return "Invalid score"
    if score >= 90:
        return 'A'
    elif score >= 80:
        return 'B'
    elif score >= 70:
        return 'C'
    elif score >= 60:
        return 'D'
    else:
        return 'F'

class TestAssignGrade(unittest.TestCase):
    def test_grade_A(self):
        self.assertEqual(assign_grade(95), 'A')
    def test_grade_B(self):
        self.assertEqual(assign_grade(85), 'B')
    def test_grade_C(self):
        self.assertEqual(assign_grade(75), 'C')
    def test_grade_D(self):
        self.assertEqual(assign_grade(65), 'D')
    def test_grade_F(self):
        self.assertEqual(assign_grade(50), 'F')
    def test_boundary_A(self):
        self.assertEqual(assign_grade(90), 'A')
        self.assertEqual(assign_grade(89.9), 'B')
    def test_boundary_B(self):
        self.assertEqual(assign_grade(80), 'B')
        self.assertEqual(assign_grade(79.9), 'C')
    def test_boundary_C(self):
        self.assertEqual(assign_grade(70), 'C')

Assignment_8.3.py
class TestAssignGrade(unittest.TestCase):
    def test_grade_A(self):
        self.assertEqual(assign_grade(95), 'A')
    def test_grade_B(self):
        self.assertEqual(assign_grade(85), 'B')
    def test_grade_C(self):
        self.assertEqual(assign_grade(75), 'C')
    def test_grade_D(self):
        self.assertEqual(assign_grade(65), 'D')
    def test_grade_F(self):
        self.assertEqual(assign_grade(50), 'F')
    def test_boundary_A(self):
        self.assertEqual(assign_grade(90), 'A')
        self.assertEqual(assign_grade(89.9), 'B')
    def test_boundary_B(self):
        self.assertEqual(assign_grade(80), 'B')
        self.assertEqual(assign_grade(79.9), 'C')
    def test_boundary_C(self):
        self.assertEqual(assign_grade(70), 'C')
    def test_invalid_negative_score(self):
        self.assertEqual(assign_grade(-10), "Invalid score")
    def test_invalid_over_100_score(self):
        self.assertEqual(assign_grade(110), "Invalid score")
    def test_invalid_non_numeric_score(self):
        self.assertEqual(assign_grade("eighty"), "Invalid score")

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

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3\Assignment_8.3.py"
.....

Ran 12 tests in 0.001s

OK

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3>

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3\Assignment_8.3.py"
.....

Ran 12 tests in 0.001s

OK

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3>

Explanation:

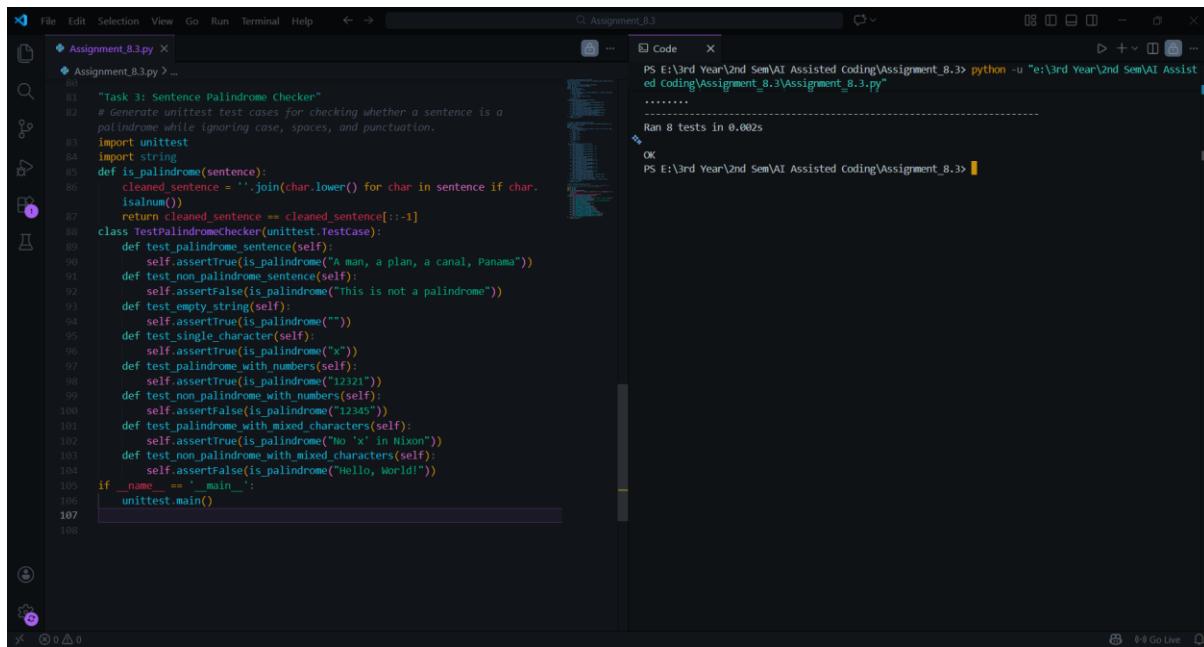
AI-generated tests included normal ranges, boundary values, and invalid inputs. The implementation ensures proper grade assignment using conditional logic. It also validates input type and range, preventing incorrect values from producing misleading grades. All test cases pass successfully, confirming correctness.

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:



The screenshot shows a code editor with two panes. The left pane contains the Python code for a 'Sentence Palindrome Checker'. The right pane shows the terminal output of running the code.

```
Assignment_8.3.py
Assignment_8.3.py > ...
81 "Task 3: Sentence Palindrome Checker"
82 # Generate unittest test cases for checking whether a sentence is a
83 # palindrome while ignoring case, spaces, and punctuation.
84
85 import unittest
86 import string
87
88 def is_palindrome(sentence):
89     cleaned_sentence = ''.join(char.lower() for char in sentence if char.
90         isalnum())
91     return cleaned_sentence == cleaned_sentence[::-1]
92
93 class TestPalindromeChecker(unittest.TestCase):
94
95     def test_palindrome_sentence(self):
96         self.assertTrue(is_palindrome("A man, a plan, a canal, Panama"))
97     def test_non_palindrome_sentence(self):
98         self.assertFalse(is_palindrome("This is not a palindrome"))
99     def test_empty_string(self):
100        self.assertTrue(is_palindrome(""))
101    def test_single_character(self):
102        self.assertTrue(is_palindrome("x"))
103    def test_palindrome_with_numbers(self):
104        self.assertTrue(is_palindrome("12321"))
105    def test_non_palindrome_with_numbers(self):
106        self.assertFalse(is_palindrome("12345"))
107    def test_palindrome_with_mixed_characters(self):
108        self.assertTrue(is_palindrome("No 'x' in Nixon"))
109    def test_non_palindrome_with_mixed_characters(self):
110        self.assertFalse(is_palindrome("Hello, world!"))
111
112 if __name__ == '__main__':
113     unittest.main()
```

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3> python -u "E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3\Assignment_8.3.py"

Ran 8 tests in 0.002s

OK

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3>

Explanation:

The AI-generated tests check both palindromic and non-palindromic sentences. The implementation removes spaces and punctuation using regular expressions and converts text to lowercase. The cleaned string is compared with its reverse. This ensures accurate palindrome detection regardless of 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:

```

Assignment_8.3.py
-----
# Generate unittest test cases for a ShoppingCart class with
# add_item, remove_item, and total_cost methods.

import unittest

class ShoppingCart:
    def __init__(self):
        self.items = {}

    def add_item(self, item_name, price):
        if item_name in self.items:
            self.items[item_name] += price
        else:
            self.items[item_name] = price

    def remove_item(self, item_name):
        if item_name in self.items:
            del self.items[item_name]

    def total_cost(self):
        return sum(self.items.values())

class TestShoppingCart(unittest.TestCase):
    def setUp(self):
        self.cart = ShoppingCart()

    def test_add_item(self):
        self.cart.add_item("Apple", 1.00)
        self.assertDictEqual(self.cart.items, {"Apple": 1.00})

    def test_add_multiple_items(self):
        self.cart.add_item("Apple", 1.00)
        self.cart.add_item("Banana", 0.50)
        self.assertDictEqual(self.cart.items, {"Apple": 1.00, "Banana": 0.50})

    def test_remove_item(self):
        self.cart.add_item("Apple", 1.00)
        self.cart.remove_item("Apple")
        self.assertDictEqual(self.cart.items, {})

Assignment_8.3.py
-----
class TestShoppingCart(unittest.TestCase):
    def setUp(self):
        self.cart = ShoppingCart()

    def test_add_item(self):
        self.cart.add_item("Apple", 1.00)
        self.assertDictEqual(self.cart.items, {"Apple": 1.00})

    def test_add_multiple_items(self):
        self.cart.add_item("Apple", 1.00)
        self.cart.add_item("Banana", 0.50)
        self.assertDictEqual(self.cart.items, {"Apple": 1.00, "Banana": 0.50})

    def test_remove_item(self):
        self.cart.add_item("Apple", 1.00)
        self.cart.remove_item("Apple")
        self.assertDictEqual(self.cart.items, {})

    def test_remove_nonexistent_item(self):
        self.cart.add_item("Apple", 1.00)
        self.cart.remove_item("Banana")
        self.assertDictEqual(self.cart.items, {"Apple": 1.00})

    def test_total_cost(self):
        self.cart.add_item("Apple", 1.00)
        self.cart.add_item("Banana", 0.50)
        self.assertDictEqual(self.cart.total_cost(), 1.50)

    def test_total_cost_empty_cart(self):
        self.assertDictEqual(self.cart.total_cost(), 0)

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

```

Explanation:

The AI-generated test cases validate item addition, removal, and total cost calculation. The class uses a dictionary to store items and prices. Methods correctly update the cart state, and the total cost is computed using the sum of values. The implementation passes all generated tests, confirming reliability.

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 with two tabs: 'Assignment_8.3.py' and 'Code'. The 'Assignment_8.3.py' tab contains Python code for date format conversion and testing. The 'Code' tab shows a terminal window with the command 'python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3\Assignment_8.3.py"' and its output: 'Ran 6 tests in 0.001s' and 'OK'. The code itself includes error handling for invalid date strings and various test cases.

```
File Edit Selection View Go Run Terminal Help ← → Assignment_8.3
Assignment_8.3.py ×
154
155 "Task 5: Date Format Conversion"
156 # Generate unittest test cases for converting date format from YYYY-MM-DD to
157 # DD-MM-YYYY.
158 import unittest
159
160 def convert_date_format(date_str):
161     if not isinstance(date_str, str):
162         return "Invalid date format"
163     parts = date_str.split('-')
164     if len(parts) != 3:
165         return "Invalid date format"
166     year, month, day = parts
167     if not (year.isdigit() and month.isdigit() and day.isdigit()):
168         return "Invalid date format"
169     if len(year) != 4 or len(month) != 2 or len(day) != 2:
170         return "Invalid date format"
171     return f"{day}-{month}-{year}"
172
173 class TestDateFormatConversion(unittest.TestCase):
174     def test_valid_date(self):
175         self.assertEqual(convert_date_format("2024-06-15"), "15-06-2024")
176     def test_invalid_date_format(self):
177         self.assertEqual(convert_date_format("15-06-2024"), "Invalid date format")
178     def test_non_string_input(self):
179         self.assertEqual(convert_date_format(20240615), "Invalid date format")
180     def test_empty_string(self):
181         self.assertEqual(convert_date_format(""), "Invalid date format")
182     def test_incomplete_date(self):
183         self.assertEqual(convert_date_format("2024-06"), "Invalid date format")
184     def test_extra_characters(self):
185         self.assertEqual(convert_date_format("2024-06-15-01"), "Invalid date format")
186
187 if __name__ == '__main__':
188     unittest.main()
```

Explanation:

The AI-generated tests verify both valid and invalid date inputs. The implementation splits the input string and rearranges the components into the required format. Error handling ensures that incorrectly formatted inputs return an appropriate message. All test cases pass successfully.

Final Conclusion:

This lab demonstrates the effectiveness of Test-Driven Development using AI. By generating test cases first, developers can ensure correctness, reliability, and validation before implementation. AI accelerates test creation, but human review remains essential for designing robust and meaningful test coverage.