

AI Assisted Coding

Assignment 8.3

Name: P. Vineeth Kumar

Hall ticket no: 2303A51256

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 tabs: 'Assignment_8.3.py' and 'Code'. The 'Assignment_8.3.py' tab contains Python code for email validation. The 'Code' tab shows the terminal output of running the script.

```
Assignment_8.3.py
1  # 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
10 import unittest
11 def is_valid_email(email):
12     if email.count('@') != 1:
13         return False
14     if email.startswith('@') or email.endswith('@') or email.startswith('.') or email.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("@test@example.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 Sem\AI Assisted Coding\Assignment_8.3\Assignment_8.3.py"
.....
Run 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:

The image shows two side-by-side code editors, both titled "Assignment_8.3.py".

Code Editor 1 (Top):

```
File Edit Selection View Go Run Terminal Help ← →
Assignment_8.3.py X
Assignment_8.3.py ->
33  "Task 2: Grade Assignment using Loops"
34  # Generate unittest test cases for a function assign_grade(score) with boundary and
35  # invalid inputs.
36  import unittest
37  def assign_grade(score):
38      if not isinstance(score, (int, float)) or score < 0 or score > 100:
39          return "Invalid score"
40      if score >= 90:
41          return 'A'
42      elif score >= 80:
43          return 'B'
44      elif score >= 70:
45          return 'C'
46      elif score >= 60:
47          return 'D'
48      else:
49          return 'F'
50  class TestAssignGrade(unittest.TestCase):
51      def test_grade_A(self):
52          self.assertEqual(assign_grade(95), 'A')
53      def test_grade_B(self):
54          self.assertEqual(assign_grade(85), 'B')
55      def test_grade_C(self):
56          self.assertEqual(assign_grade(75), 'C')
57      def test_grade_D(self):
58          self.assertEqual(assign_grade(65), 'D')
59      def test_grade_F(self):
60          self.assertEqual(assign_grade(50), 'F')
61      def test_boundary_A(self):
62          self.assertEqual(assign_grade(90), 'A')
63          self.assertEqual(assign_grade(89.9), 'B')
64      def test_boundary_B(self):
65          self.assertEqual(assign_grade(80), 'B')
66          self.assertEqual(assign_grade(79.9), 'C')
67      def test_boundary_C(self):
68          self.assertEqual(assign_grade(70), 'C')
```

Code Editor 2 (Bottom):

```
File Edit Selection View Go Run Terminal Help ← →
Assignment_8.3.py X
Assignment_8.3.py ->
49  class TestAssignGrade(unittest.TestCase):
50      def test_grade_A(self):
51          self.assertEqual(assign_grade(95), 'A')
52      def test_grade_B(self):
53          self.assertEqual(assign_grade(85), 'B')
54      def test_grade_C(self):
55          self.assertEqual(assign_grade(75), 'C')
56      def test_grade_D(self):
57          self.assertEqual(assign_grade(65), 'D')
58      def test_grade_F(self):
59          self.assertEqual(assign_grade(50), 'F')
60      def test_boundary_A(self):
61          self.assertEqual(assign_grade(90), 'A')
62          self.assertEqual(assign_grade(89.9), 'B')
63      def test_boundary_B(self):
64          self.assertEqual(assign_grade(80), 'B')
65          self.assertEqual(assign_grade(79.9), 'C')
66      def test_boundary_C(self):
67          self.assertEqual(assign_grade(70), 'C')
68          self.assertEqual(assign_grade(69.9), 'D')
69      def test_boundary_D(self):
70          self.assertEqual(assign_grade(60), 'D')
71          self.assertEqual(assign_grade(59.9), 'F')
72      def test_invalid_negative_score(self):
73          self.assertEqual(assign_grade(-10), "Invalid score")
74      def test_invalid_over_100_score(self):
75          self.assertEqual(assign_grade(110), "Invalid score")
76      def test_invalid_non_numeric_score(self):
77          self.assertEqual(assign_grade("eighty"), "Invalid score")
78  if __name__ == "__main__":
79      unittest.main()
```

Output Window (Top):

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

Output Window (Bottom):

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3> python -u "e:\3rd Year\2nd Se
ed 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 tabs: 'Assignment_8.3.py' and 'Code'. The 'Assignment_8.3.py' tab contains Python code for a sentence palindrome checker. The 'Code' tab shows the output of running the script in a terminal window. The terminal output indicates that 8 tests were run in 0.002 seconds and that all tests passed ('OK').

```
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 import unittest
85 import string
86 def is_palindrome(sentence):
87     cleaned_sentence = ''.join(char.lower() for char in sentence if char.
88     isalnum())
89     return cleaned_sentence == cleaned_sentence[::-1]
90 class TestPalindromeChecker(unittest.TestCase):
91     def test_palindrome_sentence(self):
92         self.assertTrue(is_palindrome("A man, a plan, a canal, Panama"))
93     def test_non_palindrome_sentence(self):
94         self.assertFalse(is_palindrome("This is not a palindrome"))
95     def test_empty_string(self):
96         self.assertTrue(is_palindrome(""))
97     def test_single_character(self):
98         self.assertTrue(is_palindrome("x"))
99     def test_palindrome_with_numbers(self):
100        self.assertTrue(is_palindrome("12321"))
101    def test_non_palindrome_with_numbers(self):
102        self.assertFalse(is_palindrome("12345"))
103    def test_palindrome_with_mixed_characters(self):
104        self.assertTrue(is_palindrome("No 'x' in Nixon"))
105    def test_non_palindrome_with_mixed_characters(self):
106        self.assertFalse(is_palindrome("Hello, World!"))
107 if __name__ == '__main__':
108     unittest.main()

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 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
Assignment_8.3.py > ...
111 # Generate unittest test cases for a ShoppingCart class with
112 # add_item, remove_item, and total_cost methods.
113
114 import unittest
115
116 class ShoppingCart:
117     def __init__(self):
118         self.items = {}
119     def add_item(self, item_name, price):
120         if item_name in self.items:
121             self.items[item_name] += price
122         else:
123             self.items[item_name] = price
124     def remove_item(self, item_name):
125         if item_name in self.items:
126             del self.items[item_name]
127     def total_cost(self):
128         return sum(self.items.values())
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153

```

```

Assignment_8.3.py
Assignment_8.3.py > ...
126
127 class TestShoppingCart(unittest.TestCase):
128     def setUp(self):
129         self.cart = ShoppingCart()
130     def test_add_item(self):
131         self.cart.add_item("Apple", 1.00)
132         self.assertEqual(self.cart.items, {"Apple": 1.00})
133     def test_add_multiple_items(self):
134         self.cart.add_item("Apple", 1.00)
135         self.cart.add_item("Banana", 0.50)
136         self.assertEqual(self.cart.items, {"Apple": 1.00, "Banana": 0.50})
137     def test_remove_item(self):
138         self.cart.add_item("Apple", 1.00)
139         self.cart.remove_item("Apple")
140         self.assertEqual(self.cart.items, {})
141     def test_remove_nonexistent_item(self):
142         self.cart.add_item("Apple", 1.00)
143         self.cart.remove_item("Banana")
144         self.assertEqual(self.cart.items, {"Apple": 1.00})
145     def test_total_cost(self):
146         self.cart.add_item("Apple", 1.00)
147         self.cart.add_item("Banana", 0.50)
148         self.assertEqual(self.cart.total_cost(), 1.50)
149     def test_total_cost_empty_cart(self):
150         self.assertEqual(self.cart.total_cost(), 0)
151 if __name__ == '__main__':
152     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 output of a command to run the script.

```
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()

Code
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 6 tests in 0.001s
OK
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_8.3>
```

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.