AI Assistant Coding

Assignment 8.3

Name : Sahasra**HT. No :** 2303A52499 **Batch:** 50

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

Task 1: Email Validation using TDD

Scenario

You are developing a user registration system that requires reliable email input validation.

Requirements

• **Must contain @ and . characters**

• **Must not start or end with special characters**

• **Should not allow multiple @ symbols**

• **AI should generate test cases covering valid and invalid email formats**

• **Implement is_valid_email(email) to pass all AI-generated test cases**

Expected Output

• **Python function for email validation**

• **All AI-generated test cases pass successfully**

• **Invalid email formats are correctly rejected**

• **Valid email formats return True**

#PROMPT

Generate python code for for email validation which Must contain @ and . characters and Must not start or end with special characters ,Should not allow multiple @ symbols and

generate test cases covering valid and invalid email formats .Implement
is_valid_email(email) to pass all generated test cases.

#CODE


```
 1  #2303A52503
 2  #generate python code for  for email validation  which Must contain @ and . characters  and Must not start or end with special characters ,Shoul
 3  # Implement is_valid_email(email) to pass all generated test cases
 4  import re
 5  def is_valid_email(email):
 6      # Check if email contains exactly one @ symbol
 7      if email.count('@') != 1:
 8          return False
 9
10      # Check if email contains at least one . character after the @ symbol
11      if '.' not in email.split('@')[1]:
12          return False
13
14      # Check if email starts or ends with special characters
15      if re.match(r'^[^a-zA-Z0-9]+|[^a-zA-Z0-9]+$', email):
16          return False
17
18      return True
19  # Test cases
20  test_cases = [
21      ("valid@example.com", True),
22      ("invalid.email", False),
23      ("@example.com", False),
24      ("user@", False),
25      ("user@domain.", False),
26      ("user@@domain.com", False),
27      ("user@domain.c", False),
28      (" user@example.com ", False),
29      ("user@example.com ", False),
30      (" user@example.com", False)
31  ]
32  for email, expected in test cases:
```

#OUTPUT


```
Testing 'valid@example.com': Expected True, Got True, PASS
Testing 'invalid.email': Expected False, Got False, PASS
Testing '@example.com': Expected False, Got False, PASS
Testing 'user@': Expected False, Got False, PASS
Testing 'user@domain.': Expected False, Got True, FAIL
Testing 'user@@domain.com': Expected False, Got False, PASS
Testing 'user@domain.c': Expected False, Got True, FAIL
Testing ' user@example.com ': Expected False, Got False, PASS
Testing 'user@example.com ': Expected False, Got True, FAIL
Testing ' user@example.com': Expected False, Got False, PASS
```

#Why This Prompt Was chosen

This prompt was chosen because it clearly specifies all validation rules upfront, helping the AI
generate both correct logic and meaningful test cases rather than random examples. It also
aligns with best practices in software testing by ensuring that the function is verified against
both valid and invalid scenarios.

Task 2: Grade Assignment using Loops

Scenario

You are building an automated grading system for an online examination platform.

Requirements

• **AI should generate test cases for assign_grade(score) where:**

– **90–100 → A**

– **80–89 → B**

– **70–79 → C**

– **60–69 → D**

– **Below 60 → F**

• **Include boundary values (60, 70, 80, 90)**

• **Include invalid inputs such as -5, 105, "eighty"**

• **Implement the function using a test-driven approach**

Expected Output

• **Grade assignment function implemented in Python**

• **Boundary values handled correctly**

• **Invalid inputs handled gracefully**

• **All AI-generated test cases pass**

#PROMPT

Automated grading system for an online examination platform.generate test cases for assign_grade(score) where:90–100 → A 80–89 → B 70–79 → C 60–69 → D. Below 60 → F Include boundary values (60, 70, 80, 90) Include invalid inputs such as -5, 105, "eighty" Handle boundary values and invalid inputs correctly in the implementation of assign_grade(score). Implement the function using a test-driven approach.

#CODE

```
'''automated grading system for an online examination platform.generate test cases for assign_grade(score) where:90-100 → A 80-89 → B 70-79 → C 60-69 → D
Below 60 → F Include boundary values (60, 70, 80, 90) Include invalid inputs such as -5, 105, "eighty" Handle boundary values and invalid inputs correctly in the implementation of assign_grade(score)
Implement the function using a test-driven approach'''
def assign_grade(score):
    if not isinstance(score, (int, float)):
        return "Invalid input"
    if score < 0 or score > 100:
        return "Invalid input"
    if score >= 90:
        return 'A'
    elif score >= 80:
        return 'B'
    elif score >= 70:
        return 'C'
    elif score >= 60:
        return 'D'
    else:
        return 'F'
# Test cases
test_cases = [
    (95),
    (85),
    (75),
    (65),
    (55),
    (60),  # Boundary value
    (70),  # Boundary value
    (80),  # Boundary value
    (90),  # Boundary value
    (-5),  # Invalid input
    (105),  # Invalid input
    ("eighty")  # Invalid input
]
for score in test_cases:
    result = assign_grade(score)
    print(f"Testing score '{score}': Got '{result}'")
```

#OUTPUT

```
Testing    user@example.com : Expected False, Got False, PASS
Testing score '95': Got 'A'
Testing score '85': Got 'B'
Testing score '75': Got 'C'
Testing score '65': Got 'D'
Testing score '55': Got 'F'
Testing score '60': Got 'D'
Testing score '70': Got 'C'
Testing score '80': Got 'B'
Testing score '90': Got 'A'
Testing score '-5': Got 'Invalid input'
Testing score '105': Got 'Invalid input'
Testing score 'eighty': Got 'Invalid input'
```

Why This Prompt Was Chosen

This prompt was chosen because it explicitly defines grading rules, boundary conditions, and invalid inputs, which are critical for building a robust and error-resistant function. It also promotes a test-driven approach, ensuring the implementation is validated systematically before finalizing the logic.

Task 3: Sentence Palindrome Checker

Scenario

You are developing a text-processing utility to analyze sentences.

Requirements

• **AI should generate test cases for is_sentence_palindrome(sentence)**

• **Ignore case, spaces, and punctuation**

- **Test both palindromic and non-palindromic sentences**

- **Example:**

– **"A man a plan a canal Panama" → True**

Expected Output

- **Function correctly identifies sentence palindromes**

- **Case and punctuation are ignored**

- **Returns True or False accurately**

- **All AI-generated test cases pass.**

#PROMPT

Implement a python function Sentence Palindrome Checker.Function should correctly identify sentence palindromes and Case and punctuation are ignored . It should Return True or False accurately.generate test cases for is_sentence_palindrome(sentence) Ignore case, spaces, and punctuation Test both palindromic and non-palindromic sentences.

#CODE

```python
''' Implement a python function Sentence Palindrome Checker.Function should correctly identify sentence palindromes and Case and punctuation are
import re
def is_sentence_palindrome(sentence):
    # Remove non-alphanumeric characters and convert to lowercase
    cleaned_sentence = re.sub(r'[^A-Za-z0-9]', '', sentence).lower()
    # Check if the cleaned sentence is equal to its reverse
    return cleaned_sentence == cleaned_sentence[::-1]
# Test cases
test_cases = [
    ("A man, a plan, a canal, Panama"),
    ("No 'x' in Nixon"),
    ("Was it a car or a cat I saw?"),
    ("Not a palindrome"),
    ("12321"),
    ("12345"),
    (""),  # Empty string is a palindrome
    (" "),  # String with only spaces is a palindrome
    ("!@#$%^&*()")  # String with only punctuation is a palindrome
]
for sentence in test_cases:
    result = is_sentence_palindrome(sentence)
    print(f"Testing '{sentence}': {result}")
```

#OUTPUT

```
Testing 'A man, a plan, a canal, Panama': True
Testing 'No 'x' in Nixon': True
Testing 'Was it a car or a cat I saw?': True
Testing 'Not a palindrome': False
Testing '12321': True
Testing '12345': False
Testing '': True
Testing ' ': True
Testing '!@#$%^&*()': True
```

#Why This Prompt Was Chosen

This prompt was chosen because it clearly states the normalization rules (ignoring case, spaces, and punctuation), which are essential for correctly detecting sentence-level

palindromes rather than simple word matches. It also requires both positive and negative test cases, ensuring the function is thoroughly validated and not tailored to just one example.

Task 4: ShoppingCart Class

Scenario

You are designing a basic shopping cart module for an e-commerce application.

Requirements

• **AI should generate test cases for the ShoppingCart class**

• **Class must include the following methods:**

– **add_item(name, price)**

– **remove_item(name)**

– **total_cost()**

• **Validate correct addition, removal, and cost calculation**

• **Handle empty cart scenarios**

Expected Output

• **Fully implemented ShoppingCart class**

• **All methods pass AI-generated test cases**

• **Total cost is calculated accurately**

• **Items are added and removed correctly #PROMPT**

generate a basic shopping cart module and test cases for the ShoppingCart class Class must include the following methods:

– add_item(name, price)

– remove_item(name)

– total_cost() Validate correct addition, removal, and cost calculation and Handle empty cart scenarios ,Total cost is calculated accurately Items are added and removed correctly'''

#CODE

```python
101  class ShoppingCart:
102      def __init__(self):
103          self.items = {}
104
105      def add_item(self, name, price):
106          if name in self.items:
107              self.items[name] += price
108          else:
109              self.items[name] = price
110
111      def remove_item(self, name):
112          if name in self.items:
113              del self.items[name]
114
115      def total_cost(self):
116          return sum(self.items.values())
117  # Test cases
118  cart = ShoppingCart()
119  cart.add_item("Apple", 1.00)
120  cart.add_item("Banana", 0.50)
121  print(f"Total cost after adding items: {cart.total_cost()}")  # Expected: 1.50
122  cart.add_item("Apple", 1.00)
123  print(f"Total cost after adding another Apple: {cart.total_cost()}")  # Expected: 2.50
124  cart.remove_item("Banana")
125  print(f"Total cost after removing Banana: {cart.total_cost()}")  # Expected: 2.00
126  cart.remove_item("Orange")  # Removing an item that doesn't exist should not affect the total
127  print(f"Total cost after trying to remove non-existent item: {cart.total_cost()}")  # Expected: 2.00
128  cart.remove_item("Apple")
129  print(f"Total cost after removing all Apples: {cart.total_cost()}")  # Expected: 0.00
```

#OUTPUT

```
Total cost after adding items: 1.5
Total cost after adding another Apple: 2.5
Total cost after removing Banana: 2.0
Total cost after trying to remove non-existent item: 2.0
Total cost after removing all Apples: 0
```

Task 5: Date Format Conversion

Scenario

You are creating a utility function to convert date formats for reports.

Requirements

• AI should generate test cases for convert_date_format(date_str)

• Input format must be "YYYY-MM-DD" • Output format must be "DD-MM-YYYY"

• Example:

– "2023-10-15" → "15-10-2023"

Expected Output

• Date conversion function implemented in Python

• Correct format conversion for all valid inputs

• All AI-generated test cases pass successfully

#PROMPT

'''Implement a date conversion python function to convert date formats for reports.generate test cases for convert_date_format(date_str) Input format must be "YYYY-MM-DD" and Output format must be "DD-MM-YYYY"'''

#CODE

```
131    '''Implement a date conversion python function to convert date formats for reports.generate test cases for convert_date_format(date_str) Input
132    def convert_date_format(date_str):
133        try:
134            year, month, day = date_str.split('-')
135            return f"{day}-{month}-{year}"
136        except ValueError:
137            return "Invalid date format"
138    # Test cases
139    test_cases = [
140        "2023-12-25",
141        "2020-01-01",
142        "1999-06-30",
143        "date",
144        "2023/12/25",  # Invalid format
145        "2023-13-45",  # Invalid month and day
146        "2023-12-31"
147    ]
148
149    for date in test_cases:
150        result = convert_date_format(date)
151        print(f"Converting '{date}': {result}")
```

#OUTPUT

```
● Converting '2023-12-25': 25-12-2023
  Converting '2020-01-01': 01-01-2020
  Converting '1999-06-30': 30-06-1999
  Converting 'date': Invalid date format
  Converting '2023/12/25': Invalid date format
  Converting '2023-13-45': 45-13-2023
  Converting '2023-12-31': 31-12-2023
```

#Why This Prompt Was Chosen

The original loop never ended because i was never updated, so the condition i < 5 was always true. Adding i += 1 ensures that i increases each time, allowing the loop to eventually stop when i reaches 5.