LAB ASSIGNMNET 8.3

Subject : AI Assistant Coding

Hall Ticket No : 2303A52465

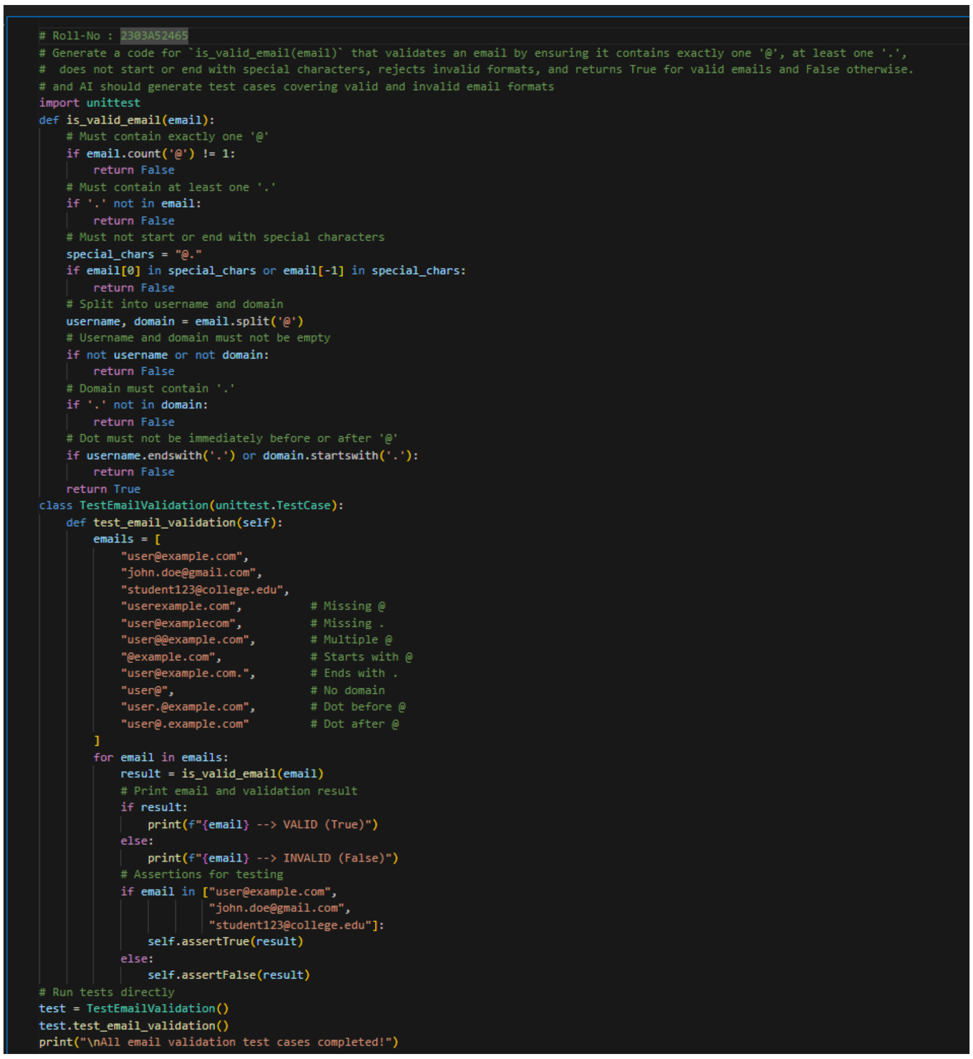
Name : M.Avinash

Batch : 50

Date : 11/2/26

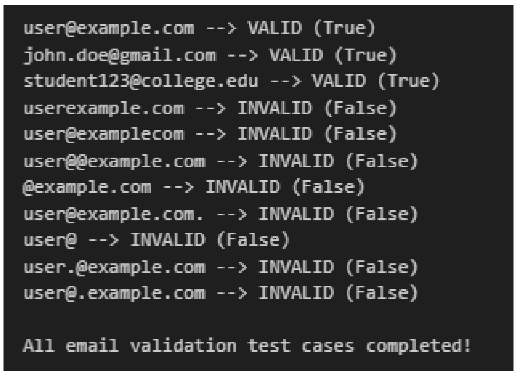
Task-1 :

Prompt : Generate a code for `is_valid_email(email)` that validates an email by ensuring it contains exactly one '@', at least one '.', does not start or end with special characters, rejects invalid formats, and returns True for valid emails and False otherwise. and AI should generate test cases covering valid and invalid email formats



```python
# Roll-No : 2303A52465
# Generate a code for `is_valid_email(email)` that validates an email by ensuring it contains exactly one '@', at least one '.',
#  does not start or end with special characters, rejects invalid formats, and returns True for valid emails and False otherwise.
# and AI should generate test cases covering valid and invalid email formats
import unittest
def is_valid_email(email):
    # Must contain exactly one '@'
    if email.count('@') != 1:
        return False
    # Must contain at least one '.'
    if '.' not in email:
        return False
    # Must not start or end with special characters
    special_chars = "@."
    if email[0] in special_chars or email[-1] in special_chars:
        return False
    # Split into username and domain
    username, domain = email.split('@')
    # Username and domain must not be empty
    if not username or not domain:
        return False
    # Domain must contain '.'
    if '.' not in domain:
        return False
    # Dot must not be immediately before or after '@'
    if username.endswith('.') or domain.startswith('.'):
        return False
    return True
class TestEmailValidation(unittest.TestCase):
    def test_email_validation(self):
        emails = [
            "user@example.com",
            "john.doe@gmail.com",
            "student123@college.edu",
            "userexample.com",          # Missing @
            "user@examplecom",          # Missing .
            "user@@example.com",        # Multiple @
            "@example.com",             # Starts with @
            "user@example.com.",        # Ends with .
            "user@",                    # No domain
            "user.@example.com",        # Dot before @
            "user@.example.com"         # Dot after @
        ]
        for email in emails:
            result = is_valid_email(email)
            # Print email and validation result
            if result:
                print(f"{email} --> VALID (True)")
            else:
                print(f"{email} --> INVALID (False)")
            # Assertions for testing
            if email in ["user@example.com",
                         "john.doe@gmail.com",
                         "student123@college.edu"]:
                self.assertTrue(result)
            else:
                self.assertFalse(result)
# Run tests directly
test = TestEmailValidation()
test.test_email_validation()
print("\nAll email validation test cases completed!")
```

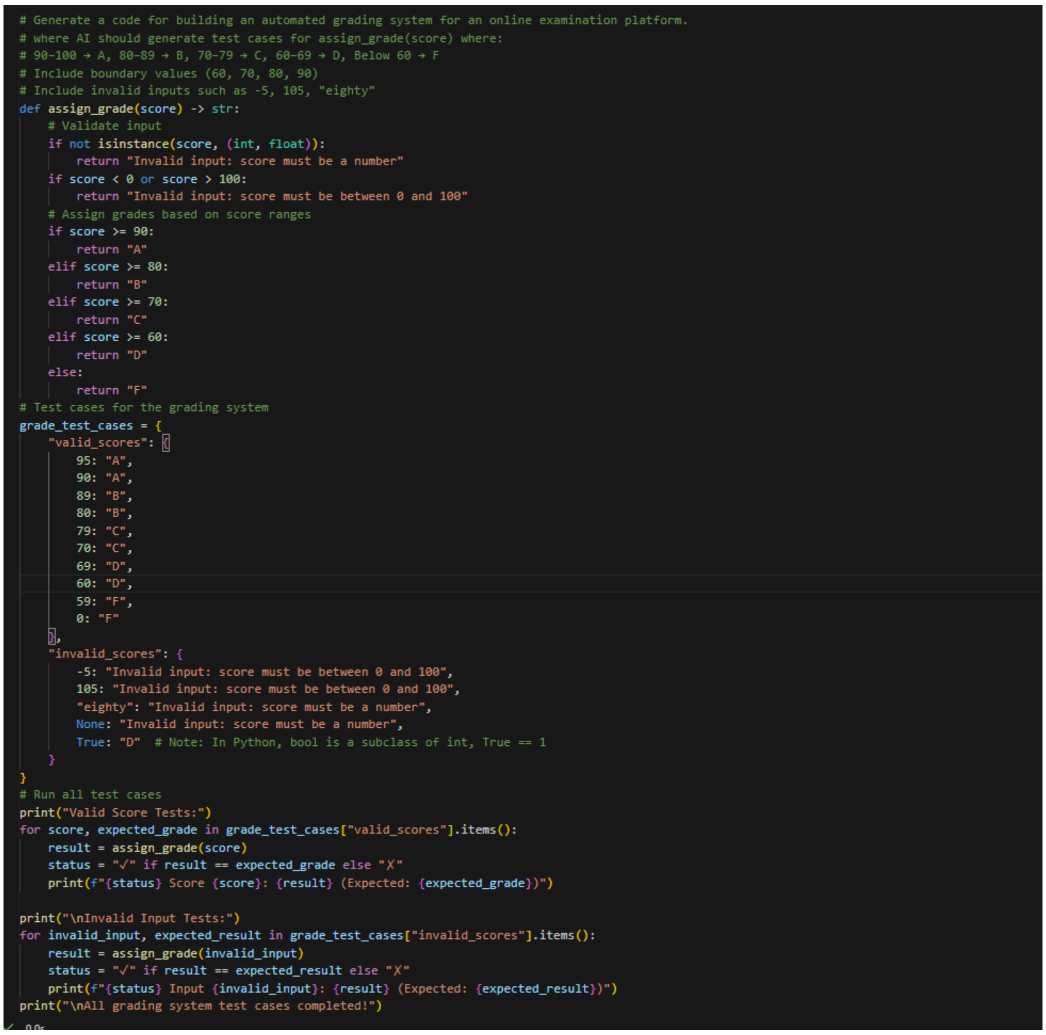
Output :



```
user@example.com --> VALID (True)
john.doe@gmail.com --> VALID (True)
student123@college.edu --> VALID (True)
userexample.com --> INVALID (False)
user@examplecom --> INVALID (False)
user@@example.com --> INVALID (False)
@example.com --> INVALID (False)
user@example.com. --> INVALID (False)
user@ --> INVALID (False)
user.@example.com --> INVALID (False)
user@.example.com --> INVALID (False)

All email validation test cases completed!
```

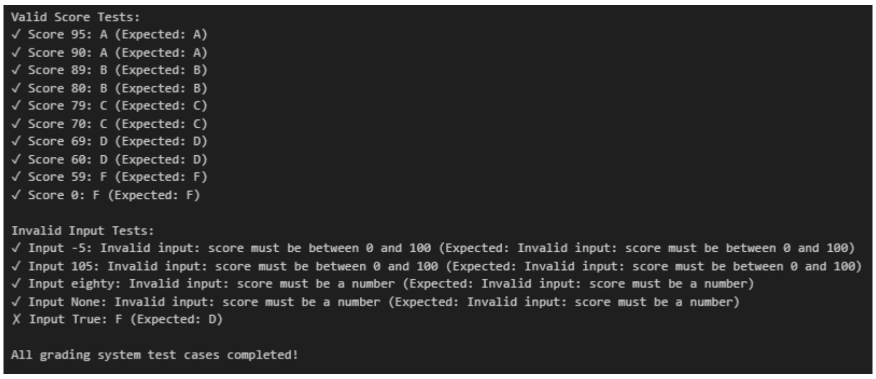
Task-2 :

Prompt : Generate a code for building an automated grading system for an online examination platform. where 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) and Include invalid inputs such as -5, 105, "eighty"



```python
# Generate a code for building an automated grading system for an online examination platform.
# where 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"
def assign_grade(score) -> str:
    # Validate input
    if not isinstance(score, (int, float)):
        return "Invalid input: score must be a number"
    if score < 0 or score > 100:
        return "Invalid input: score must be between 0 and 100"
    # Assign grades based on score ranges
    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 for the grading system
grade_test_cases = {
    "valid_scores": {
        95: "A",
        90: "A",
        89: "B",
        80: "B",
        79: "C",
        70: "C",
        69: "D",
        60: "D",
        59: "F",
        0: "F"
    },
    "invalid_scores": {
        -5: "Invalid input: score must be between 0 and 100",
        105: "Invalid input: score must be between 0 and 100",
        "eighty": "Invalid input: score must be a number",
        None: "Invalid input: score must be a number",
        True: "D"  # Note: In Python, bool is a subclass of int, True == 1
    }
}
# Run all test cases
print("Valid Score Tests:")
for score, expected_grade in grade_test_cases["valid_scores"].items():
    result = assign_grade(score)
    status = "✓" if result == expected_grade else "✗"
    print(f"{status} Score {score}: {result} (Expected: {expected_grade})")

print("\nInvalid Input Tests:")
for invalid_input, expected_result in grade_test_cases["invalid_scores"].items():
    result = assign_grade(invalid_input)
    status = "✓" if result == expected_result else "✗"
    print(f"{status} Input {invalid_input}: {result} (Expected: {expected_result})")
print("\nAll grading system test cases completed!")
```

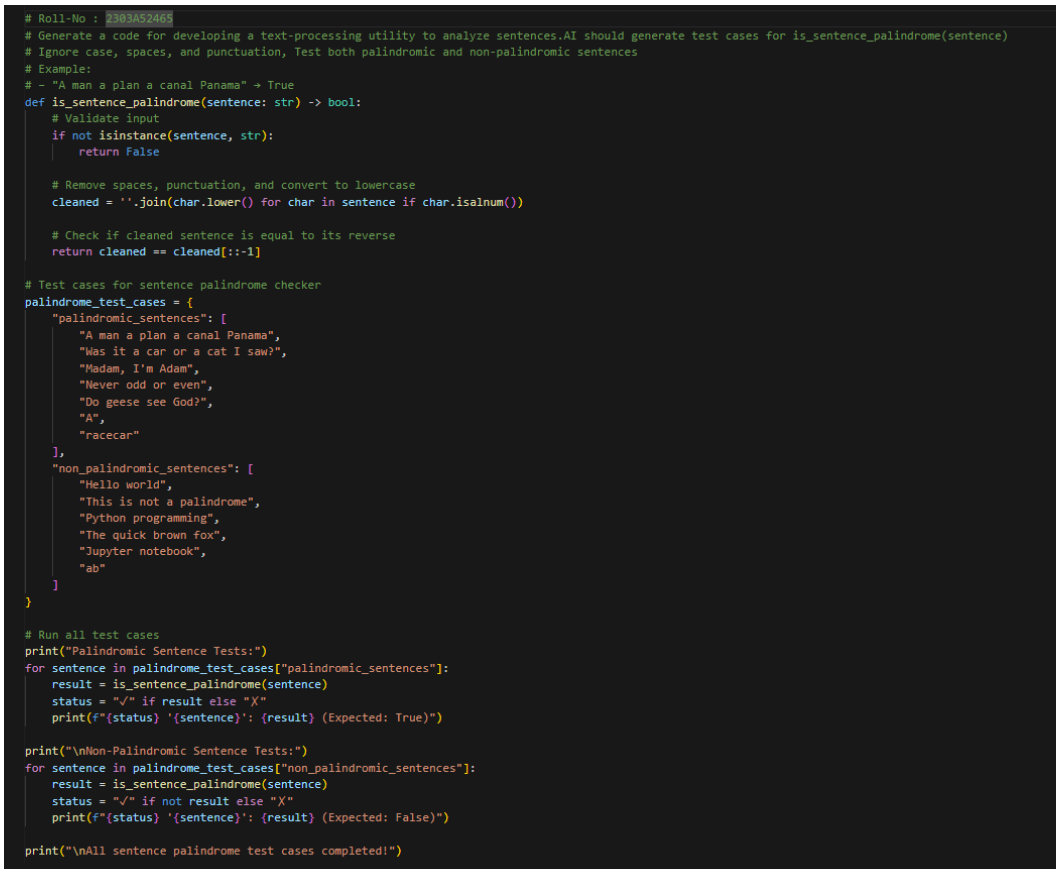
Output :



```
Valid Score Tests:
✓ Score 95: A (Expected: A)
✓ Score 90: A (Expected: A)
✓ Score 89: B (Expected: B)
✓ Score 80: B (Expected: B)
✓ Score 79: C (Expected: C)
✓ Score 70: C (Expected: C)
✓ Score 69: D (Expected: D)
✓ Score 60: D (Expected: D)
✓ Score 59: F (Expected: F)
✓ Score 0: F (Expected: F)

Invalid Input Tests:
✓ Input -5: Invalid input: score must be between 0 and 100 (Expected: Invalid input: score must be between 0 and 100)
✓ Input 105: Invalid input: score must be between 0 and 100 (Expected: Invalid input: score must be between 0 and 100)
✓ Input eighty: Invalid input: score must be a number (Expected: Invalid input: score must be a number)
✓ Input None: Invalid input: score must be a number (Expected: Invalid input: score must be a number)
✗ Input True: F (Expected: D)

All grading system test cases completed!
```

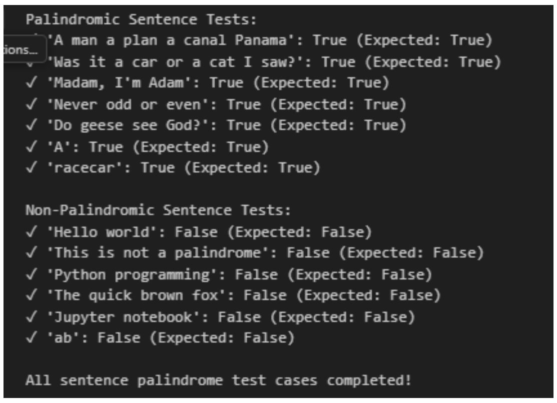
Task-3 :

Prompt : Generate a code for developing a text-processing utility to analyze sentences.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



```python
# Roll-No : 2303A52465
# Generate a code for developing a text-processing utility to analyze sentences.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
def is_sentence_palindrome(sentence: str) -> bool:
    # Validate input
    if not isinstance(sentence, str):
        return False

    # Remove spaces, punctuation, and convert to lowercase
    cleaned = ''.join(char.lower() for char in sentence if char.isalnum())

    # Check if cleaned sentence is equal to its reverse
    return cleaned == cleaned[::-1]

# Test cases for sentence palindrome checker
palindrome_test_cases = {
    "palindromic_sentences": [
        "A man a plan a canal Panama",
        "Was it a car or a cat I saw?",
        "Madam, I'm Adam",
        "Never odd or even",
        "Do geese see God?",
        "A",
        "racecar"
    ],
    "non_palindromic_sentences": [
        "Hello world",
        "This is not a palindrome",
        "Python programming",
        "The quick brown fox",
        "Jupyter notebook",
        "ab"
    ]
}

# Run all test cases
print("Palindromic Sentence Tests:")
for sentence in palindrome_test_cases["palindromic_sentences"]:
    result = is_sentence_palindrome(sentence)
    status = "✓" if result else "✗"
    print(f"{status} '{sentence}': {result} (Expected: True)")

print("\nNon-Palindromic Sentence Tests:")
for sentence in palindrome_test_cases["non_palindromic_sentences"]:
    result = is_sentence_palindrome(sentence)
    status = "✓" if not result else "✗"
    print(f"{status} '{sentence}': {result} (Expected: False)")

print("\nAll sentence palindrome test cases completed!")
```

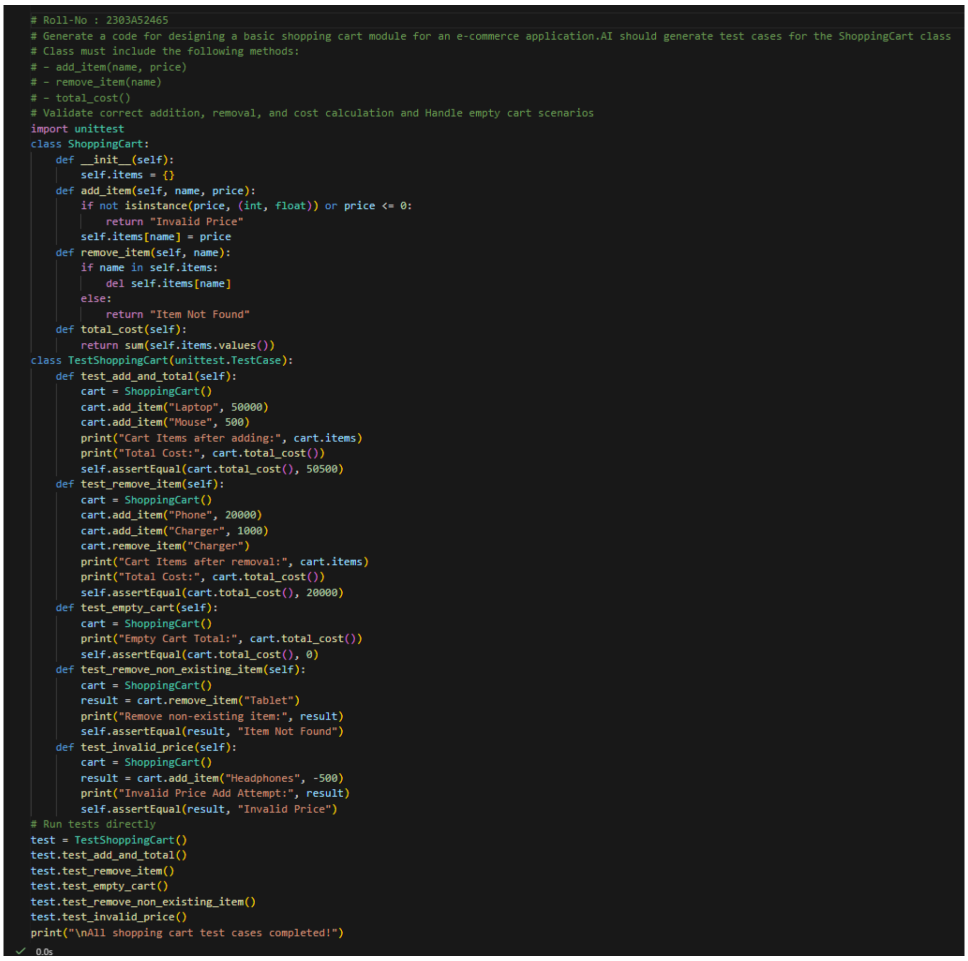
Output :



```
Palindromic Sentence Tests:
✓ 'A man a plan a canal Panama': True (Expected: True)
✓ 'Was it a car or a cat I saw?': True (Expected: True)
✓ 'Madam, I'm Adam': True (Expected: True)
✓ 'Never odd or even': True (Expected: True)
✓ 'Do geese see God?': True (Expected: True)
✓ 'A': True (Expected: True)
✓ 'racecar': True (Expected: True)

Non-Palindromic Sentence Tests:
✓ 'Hello world': False (Expected: False)
✓ 'This is not a palindrome': False (Expected: False)
✓ 'Python programming': False (Expected: False)
✓ 'The quick brown fox': False (Expected: False)
✓ 'Jupyter notebook': False (Expected: False)
✓ 'ab': False (Expected: False)

All sentence palindrome test cases completed!
```

Task-4 :

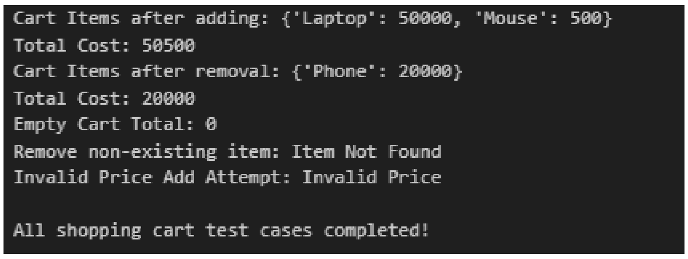
Prompt : Generate a code for designing a basic shopping cart module for an e-commerce application.AI should generate test cases for the ShoppingCart class
Class must include the following methods:
– add_item(name, price)
– remove_item(name)
– total_cost()
Validate correct addition, removal, and cost calculation and Handle empty cart scenarios



```python
# Roll-No : 2303A52465
# Generate a code for designing a basic shopping cart module for an e-commerce application.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 and Handle empty cart scenarios
import unittest
class ShoppingCart:
    def __init__(self):
        self.items = {}
    def add_item(self, name, price):
        if not isinstance(price, (int, float)) or price <= 0:
            return "Invalid Price"
        self.items[name] = price
    def remove_item(self, name):
        if name in self.items:
            del self.items[name]
        else:
            return "Item Not Found"
    def total_cost(self):
        return sum(self.items.values())
class TestShoppingCart(unittest.TestCase):
    def test_add_and_total(self):
        cart = ShoppingCart()
        cart.add_item("Laptop", 50000)
        cart.add_item("Mouse", 500)
        print("Cart Items after adding:", cart.items)
        print("Total Cost:", cart.total_cost())
        self.assertEqual(cart.total_cost(), 50500)
    def test_remove_item(self):
        cart = ShoppingCart()
        cart.add_item("Phone", 20000)
        cart.add_item("Charger", 1000)
        cart.remove_item("Charger")
        print("Cart Items after removal:", cart.items)
        print("Total Cost:", cart.total_cost())
        self.assertEqual(cart.total_cost(), 20000)
    def test_empty_cart(self):
        cart = ShoppingCart()
        print("Empty Cart Total:", cart.total_cost())
        self.assertEqual(cart.total_cost(), 0)
    def test_remove_non_existing_item(self):
        cart = ShoppingCart()
        result = cart.remove_item("Tablet")
        print("Remove non-existing item:", result)
        self.assertEqual(result, "Item Not Found")
    def test_invalid_price(self):
        cart = ShoppingCart()
        result = cart.add_item("Headphones", -500)
        print("Invalid Price Add Attempt:", result)
        self.assertEqual(result, "Invalid Price")
# Run tests directly
test = TestShoppingCart()
test.test_add_and_total()
test.test_remove_item()
test.test_empty_cart()
test.test_remove_non_existing_item()
test.test_invalid_price()
print("\nAll shopping cart test cases completed!")
```

✓ 0.0s

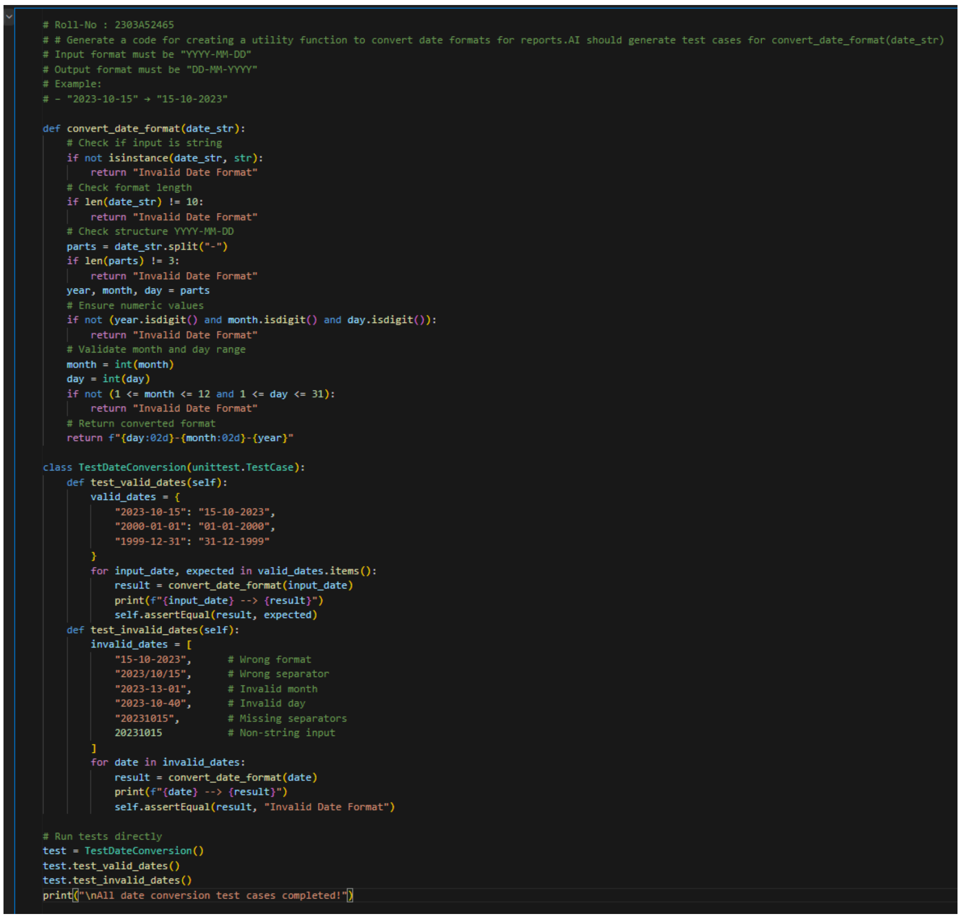
Output :



```
Cart Items after adding: {'Laptop': 50000, 'Mouse': 500}
Total Cost: 50500
Cart Items after removal: {'Phone': 20000}
Total Cost: 20000
Empty Cart Total: 0
Remove non-existing item: Item Not Found
Invalid Price Add Attempt: Invalid Price

All shopping cart test cases completed!
```

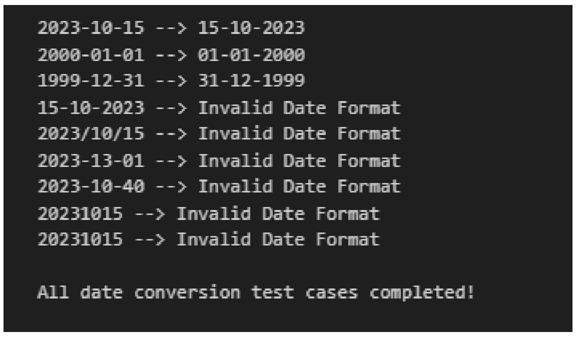
Task-5 :

Prompt : Generate a code for creating a utility function to convert date formats for reports.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"



```python
# Roll-No : 2303A52465
# # Generate a code for creating a utility function to convert date formats for reports.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"

def convert_date_format(date_str):
    # Check if input is string
    if not isinstance(date_str, str):
        return "Invalid Date Format"
    # Check format length
    if len(date_str) != 10:
        return "Invalid Date Format"
    # Check structure YYYY-MM-DD
    parts = date_str.split("-")
    if len(parts) != 3:
        return "Invalid Date Format"
    year, month, day = parts
    # Ensure numeric values
    if not (year.isdigit() and month.isdigit() and day.isdigit()):
        return "Invalid Date Format"
    # Validate month and day range
    month = int(month)
    day = int(day)
    if not (1 <= month <= 12 and 1 <= day <= 31):
        return "Invalid Date Format"
    # Return converted format
    return f"{day:02d}-{month:02d}-{year}"

class TestDateConversion(unittest.TestCase):
    def test_valid_dates(self):
        valid_dates = {
            "2023-10-15": "15-10-2023",
            "2000-01-01": "01-01-2000",
            "1999-12-31": "31-12-1999"
        }
        for input_date, expected in valid_dates.items():
            result = convert_date_format(input_date)
            print(f"{input_date} --> {result}")
            self.assertEqual(result, expected)
    def test_invalid_dates(self):
        invalid_dates = [
            "15-10-2023",      # Wrong format
            "2023/10/15",      # Wrong separator
            "2023-13-01",      # Invalid month
            "2023-10-40",      # Invalid day
            "20231015",        # Missing separators
            20231015           # Non-string input
        ]
        for date in invalid_dates:
            result = convert_date_format(date)
            print(f"{date} --> {result}")
            self.assertEqual(result, "Invalid Date Format")

# Run tests directly
test = TestDateConversion()
test.test_valid_dates()
test.test_invalid_dates()
print("\nAll date conversion test cases completed!")
```

Output :



```
2023-10-15 --> 15-10-2023
2000-01-01 --> 01-01-2000
1999-12-31 --> 31-12-1999
15-10-2023 --> Invalid Date Format
2023/10/15 --> Invalid Date Format
2023-13-01 --> Invalid Date Format
2023-10-40 --> Invalid Date Format
20231015 --> Invalid Date Format
20231015 --> Invalid Date Format

All date conversion test cases completed!
```