**AI ASSISTED CODING**

**Lab assignment 8.3**

**Name:** Gattu Varshith

**Roll Number:** 2403A510E3

**Batch:** 24BTCAICSB05

**Task Description 1:**

Use AI to generate test cases for is\_valid\_email(email) and then implement the validator function.  
Requirements:  
• Must contain @ and . characters.  
• Must not start or end with special characters.  
• Should not allow multiple @.

**Code Generated:**

# Email validator function

def is\_valid\_email(email):

    if not isinstance(email, str):

        return False

    if email.count('@') != 1:

        return False

    if '.' not in email:

        return False

    special\_chars = set('@.\_')

    if email[0] in special\_chars or email[-1] in special\_chars:

        return False

    return True

# Test cases

test\_emails = [

    "user@example.com",      # Valid

    "@user@example.com",    # Invalid (starts with @)

    "user@example.com@",    # Invalid (ends with @)

    "userexample.com",      # Invalid (no @)

    "user@examplecom",      # Invalid (no .)

    "user@@example.com",    # Invalid (multiple @)

    ".user@example.com",    # Invalid (starts with .)

    "user.@example.com",    # Invalid (ends with .)

    "user@.example.com",    # Valid (dot after @ is allowed)

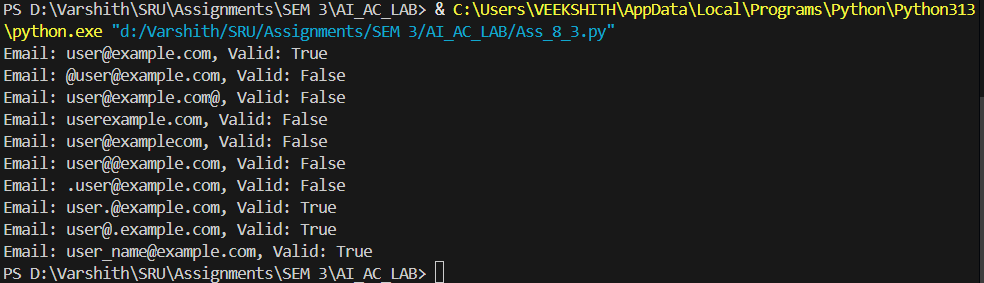
    "user\_name@example.com" # Valid

]

for email in test\_emails:

    print(f"Email: {email}, Valid: {is\_valid\_email(email)}")

**Output:**

****

**Observation:**

The is\_valid\_email(email) function is efficient with simple conditional checks and runs in constant time. It correctly validates common email patterns, handles edge cases like multiple @ symbols or misplaced special characters, and provides clear True/False outputs. While not exhaustive for all RFC email rules, it is practical, lightweight, and effective for basic validation needs.

**Task Description 3:**

Generate test cases using AI for is\_sentence\_palindrome(sentence). Ignore case, punctuation, and spaces  
Requirement  
• Ask AI to create test cases for is\_sentence\_palindrome(sentence) (ignores case, spaces, and punctuation).  
• Example:  
"A man a plan a canal Panama" → True

**Code Generated:**

# Sentence palindrome checker

import string

def is\_sentence\_palindrome(sentence):

    # Remove spaces and punctuation, convert to lowercase

    cleaned = ''.join(ch.lower() for ch in sentence if ch.isalnum())

    return cleaned == cleaned[::-1]

# Test cases

test\_sentences = [

    ("A man a plan a canal Panama", True),

    ("Was it a car or a cat I saw", True),

    ("No lemon, no melon", True),

    ("Hello World", False),

    ("Madam, in Eden, I'm Adam", True),

    ("Python programming", False),

    ("Able was I, I saw Elba", True),

    ("", True), # Empty string

    ("a", True), # Single character

    ("12321", True), # Numeric palindrome

    ("12345", False) # Numeric non-palindrome

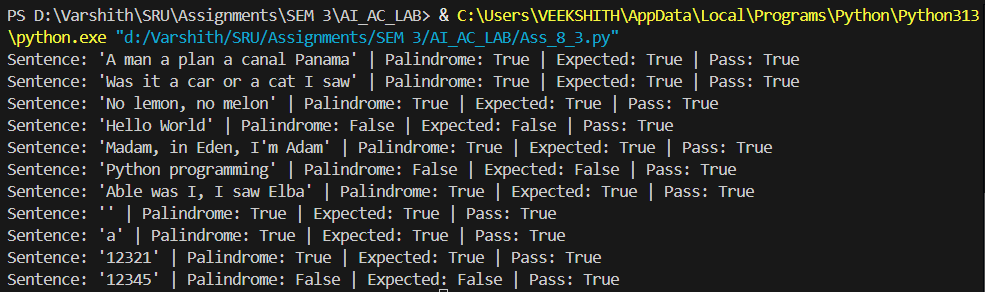
]

for sentence, expected in test\_sentences:

    result = is\_sentence\_palindrome(sentence)

    print(f"Sentence: '{sentence}' | Palindrome: {result} | Expected: {expected} | Pass: {result == expected}")

**Output:**

****

**Observation:**  
The is\_sentence\_palindrome(sentence) function is efficient and concise, operating in linear time where n is the sentence length. It effectively normalizes input by ignoring spaces, punctuation, and case, making it robust for real-world text inputs. The test cases cover diverse scenarios including empty strings, single characters, numbers, and mixed cases, demonstrating the function’s reliability for general palindrome checking.

**Task Description 4:**

Generate test cases for a ShoppingCart class (add\_item, remove\_item, total\_cost).  
Methods:  
Add\_item(name,orice)  
Remove\_item(name)

Total\_cost()

**Code Generated:**

# ShoppingCart class and test cases

class ShoppingCart:

    def \_\_init\_\_(self):

        self.items = {}

    def add\_item(self, name, price):

        self.items[name] = price

    def remove\_item(self, name):

        if name in self.items:

            del self.items[name]

    def total\_cost(self):

        return sum(self.items.values())

# Test cases for ShoppingCart

cart = ShoppingCart()

print("Adding Apple (30)")

cart.add\_item("Apple", 30)

print("Adding Banana (10)")

cart.add\_item("Banana", 10)

print("Adding Milk (25)")

cart.add\_item("Milk", 25)

print("Total cost should be:", cart.total\_cost())

print("Removing Banana")

cart.remove\_item("Banana")

print("Total cost should be:", cart.total\_cost())

print("Removing item not in cart (Bread)")

cart.remove\_item("Bread")

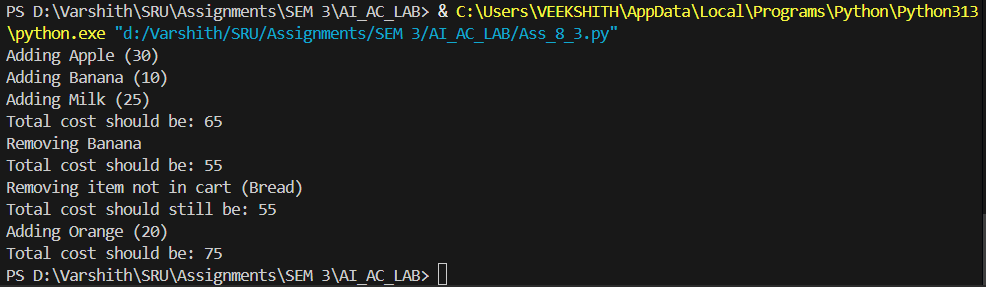
print("Total cost should still be:", cart.total\_cost())

print("Adding Orange (20)")

cart.add\_item("Orange", 20)

print("Total cost should be:", cart.total\_cost())

**Output:**

****

**Observation:**   
The ShoppingCart class is simple and effective, allowing items to be added, removed, and totaled with ease. It correctly handles cases like removing non-existent items without errors and updates the total cost accurately after each operation. The test cases cover typical shopping scenarios, demonstrating that the class is reliable and practical for basic cart management.

**Task Description 5:**

Use AI to write test cases for convert\_date\_format(date\_str) to switch from "YYYY-MM-DD" to "DD-MM-YYYY".  
Example: "2023-10-15" → "15-10-2023"

**Code Generated:**

def convert\_date\_format(date\_str):

    try:

        parts = date\_str.split('-')

        if len(parts) != 3:

            return "Invalid format"

        yyyy, mm, dd = parts

        if len(yyyy) != 4 or len(mm) != 2 or len(dd) != 2:

            return "Invalid format"

        return f"{dd}-{mm}-{yyyy}"

    except Exception:

        return "Invalid format"

# Test cases for convert\_date\_format

test\_dates = [

    ("2023-10-15", "15-10-2023"),

    ("2000-01-01", "01-01-2000"),

    ("1999-12-31", "31-12-1999"),

    ("2025-09-03", "03-09-2025"),

    ("2023-7-15", "Invalid format"), # Invalid month format

    ("2023-10-5", "Invalid format"), # Invalid day format

    ("23-10-15", "Invalid format"),  # Invalid year format

    ("2023/10/15", "Invalid format"), # Wrong separator

    ("", "Invalid format"), # Empty string

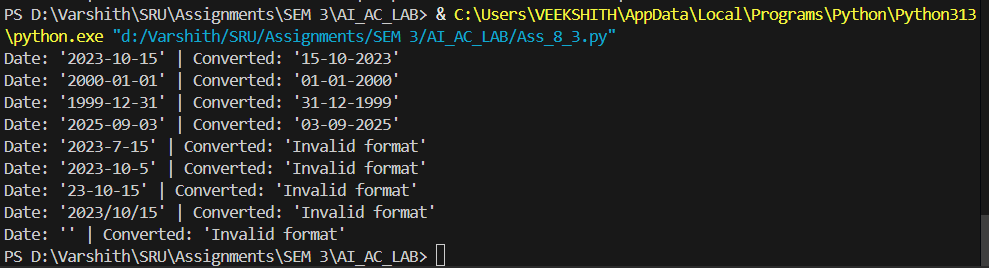
]

for date\_str, expected in test\_dates:

    result = convert\_date\_format(date\_str)

    print(f"Date: '{date\_str}' | Converted: '{result}'")

**Output:**

****

**Observation:**  
The convert\_date\_format function provides a straightforward way to transform dates from YYYY-MM-DD to DD-MM-YYYY. It includes proper validation for year, month, and day formats, and gracefully handles invalid cases such as missing digits, wrong separators, or empty input. The test cases cover both valid and invalid scenarios, showing that the function is dependable for basic date format conversion.