AI ASSISTED CODING

**ASSISGNMENT 8.3**

**2403A510B4**

**BATCH-05**

QUESTION

**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:**

def is\_valid\_email(email):

    # Check for presence of '@' and '.'

    if '@' not in email or '.' not in email:

        return False

    # Should not allow multiple '@'

    if email.count('@') != 1:

        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

    return True

# Test cases

test\_emails = [

    "user@example.com",        # Valid

    "user.name@example.com",   # Valid

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

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

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

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

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

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

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

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

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

    "user@ex.ample.com",       # Valid

]

for email in test\_emails:

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

print("output:Email validation logic implemented successfully")

**OUTPUT:**

user@example.com: True

user.name@example.com: True

user@domain: False

@user.com: False

user@.com: True

user@@example.com: False

user@example..com: True

user@example.com.: False

user@example.com@: False

userexample.com: False

.user@example.com: False

user@ex.ample.com: True

output:Email validation logic implemented successfully

**OBSERVATION:**

The given code implements a simple email validation function by checking basic rules such as the presence of @ and ., ensuring only one @ exists, and rejecting emails that start or end with special characters like ., \_, or @. While it correctly identifies many valid and invalid cases, it has some limitations, such as allowing consecutive dots in the domain part (user@example..com), which is not valid in real email standards. An improved version can fix this by splitting the email into local and domain parts, ensuring both are non-empty, checking that the domain contains at least one dot, and rejecting cases with consecutive dots. Overall, the logic works well for simple validation but is not fully compliant with strict email rules.

**Task Description#2 (Loops)**  
• Ask AI to generate test cases for assign\_grade(score) function. Handle boundary and  
invalid inputs.  
Requirements  
• AI should generate test cases for assign\_grade(score) where: 90-100: A, 80-89: B, 70-  
79: C, 60-69: D, <60: F  
• Include boundary values and invalid inputs (e.g., -5, 105, "eighty")

**CODE GENERATED:**

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\_scores = [

    100,    # A (upper boundary)

    90,     # A (lower boundary)

    89,     # B (upper boundary)

    80,     # B (lower boundary)

    79,     # C (upper boundary)

    70,     # C (lower boundary)

    69,     # D (upper boundary)

    60,     # D (lower boundary)

    59,     # F (just below D)

    0,      # F (lowest valid)

    -5,     # Invalid (negative)

    105,    # Invalid (above 100)

    "eighty", # Invalid (string)

    None,   # Invalid (NoneType)

]

for score in test\_scores:

    print(f"Score: {score} => Grade: {assign\_grade(score)}")

print("Output:\nGrade assignment function tested with various inputs.")

**OUTPUT:**

Score: 100 => Grade: A

Score: 90 => Grade: A

Score: 89 => Grade: B

Score: 80 => Grade: B

Score: 79 => Grade: C

Score: 70 => Grade: C

Score: 69 => Grade: D

Score: 60 => Grade: D

Score: 59 => Grade: F

Score: 0 => Grade: F

Score: -5 => Grade: Invalid input

Score: 105 => Grade: Invalid input

Score: eighty => Grade: Invalid input

Score: None => Grade: Invalid input

Output:

Grade assignment function tested with various inputs.

**OBSERVATION:**

The code defines a grade assignment function that validates input and assigns grades based on score ranges. It correctly handles invalid inputs such as negative numbers, values above 100, non-numeric data, and None. For valid scores, it assigns grades A through F according to standard ranges, while also covering edge cases at boundary values like 90, 80, 70, and 60. Overall, the function works effectively for both valid and invalid test cases, ensuring reliable grade classification.

**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).

**CODE GENERATED:**

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]

# AI-generated test cases

test\_sentences = [

    "A man a plan a canal Panama",              # True

    "No lemon, no melon",                       # True

    "Was it a car or a cat I saw?",             # True

    "Madam In Eden, I'm Adam",                  # True

    "Step on no pets",                          # True

    "Eva, can I see bees in a cave?",           # True

    "Red roses run no risk, sir, on Nurse's order", # True

    "Never odd or even",                        # True

    "Hello, World!",                            # False

    "Not a palindrome",                         # False

    "12321",                                    # True

    "12345",                                    # False

    "Able was I, I saw Elba",                   # True

    "Go hang a salami I'm a lasagna hog",       # True

    "Was it a car or a cat I saw",              # True

    "This is not a palindrome",                 # False

]

for sentence in test\_sentences:

    print(f"'{sentence}' -> {is\_sentence\_palindrome(sentence)}")

print("OUTPUT:\nFunction returns True/False based on whether the input sentence is a palindrome, ignoring spaces, punctuation, and case.")

**OUTPUT:**

'A man a plan a canal Panama' -> True

'No lemon, no melon' -> True

'Was it a car or a cat I saw?' -> True

'Madam In Eden, I'm Adam' -> True

'Step on no pets' -> True

'Eva, can I see bees in a cave?' -> True

'Red roses run no risk, sir, on Nurse's order' -> True

'Never odd or even' -> True

'Hello, World!' -> False

'Not a palindrome' -> False

'12321' -> True

'12345' -> False

'Able was I, I saw Elba' -> True

'Go hang a salami I'm a lasagna hog' -> True

'Was it a car or a cat I saw' -> True

'This is not a palindrome' -> False

OUTPUT:

Function returns True/False based on whether the input sentence is a palindrome, ignoring spaces, punctuation, and case.

**OBSERVATION:**

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]

# AI-generated test cases

test\_sentences = [

    "A man a plan a canal Panama",              # True

    "No lemon, no melon",                       # True

    "Was it a car or a cat I saw?",             # True

    "Madam In Eden, I'm Adam",                  # True

    "Step on no pets",                          # True

    "Eva, can I see bees in a cave?",           # True

    "Red roses run no risk, sir, on Nurse's order", # True

    "Never odd or even",                        # True

    "Hello, World!",                            # False

    "Not a palindrome",                         # False

    "12321",                                    # True

    "12345",                                    # False

    "Able was I, I saw Elba",                   # True

    "Go hang a salami I'm a lasagna hog",       # True

    "Was it a car or a cat I saw",              # True

    "This is not a palindrome",                 # False

]

for sentence in test\_sentences:

    print(f"'{sentence}' -> {is\_sentence\_palindrome(sentence)}")

print("OUTPUT:\nFunction returns True/False based on whether the input sentence is a palindrome, ignoring spaces, punctuation, and case.")

**Task Description#4**  
• Let AI fix it Prompt AI to generate test cases for a ShoppingCart class (add\_item,  
remove\_item, total\_cost).  
Methods:  
Add\_item(name,orice)  
Remove\_item(name)

Total\_cost()

**CODE GEERATED:**

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())

# AI-generated test cases

cart = ShoppingCart()

cart.add\_item("apple", 1.5)

cart.add\_item("banana", 2.0)

cart.add\_item("milk", 3.25)

print("Total after adding items:", cart.total\_cost())  # Should be 6.75

cart.remove\_item("banana")

print("Total after removing banana:", cart.total\_cost())  # Should be 4.75

cart.add\_item("bread", 2.5)

print("Total after adding bread:", cart.total\_cost())  # Should be 7.25

cart.remove\_item("apple")

print("Total after removing apple:", cart.total\_cost())  # Should be 5.75

cart.remove\_item("not\_in\_cart")  # Should do nothing

print("Total after trying to remove non-existent item:", cart.total\_cost())  # Should be 5.75

print("Output:\nFull class with tested functionalities")

**OUTPUT:**

Total after adding items: 6.75

Total after removing banana: 4.75

Total after adding bread: 7.25

Total after removing apple: 5.75

Total after trying to remove non-existent item: 5.75

Output:

Full class with tested functionalities

**OBSERVATION:**

The given code defines a simple **ShoppingCart class** that allows adding items with their prices, removing items, and calculating the total cost of all items in the cart. The test cases verify different scenarios such as adding multiple items, removing specific items, and attempting to remove an item not present in the cart. The function correctly updates the total cost after each operation, ensuring accuracy in price calculation. Overall, the class works as expected for basic shopping cart functionalities and handles both valid and invalid removal cases smoothly.

**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):

    parts = date\_str.split('-')

    if len(parts) != 3:

        return "Invalid format"

    yyyy, mm, dd = parts

    return f"{dd}-{mm}-{yyyy}"

# Test cases

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"

    "2022-02-28",    # "28-02-2022"

    "2021-07-04",    # "04-07-2021"

    "abcd-ef-gh",    # "Invalid format"

    "2023-10",       # "Invalid format"

    "2023-10-5",     # "5-10-2023"

]

for date in test\_dates:

    print(f"{date} -> {convert\_date\_format(date)}")

print("Output:\nFunction converts input format correctly for all test cases")

**OUTPUT:**

Total after adding items: 6.75

Total after adding items: 6.75

Total after removing banana: 4.75

Total after removing banana: 4.75

Total after adding bread: 7.25

Total after adding bread: 7.25

Total after removing apple: 5.75

Total after removing apple: 5.75

Total after trying to remove non-existent item: 5.75

Output:

Full class with tested functionalities

**OBSERVATION:**

The code defines a simple date conversion function that changes the format from **YYYY-MM-DD** to **DD-MM-YYYY**. It works by splitting the input string by - and checking if it contains exactly three parts. Valid inputs are correctly reformatted, while incorrectly structured dates (like missing parts or non-numeric values) return "Invalid format". The test cases demonstrate correct handling of normal dates, edge cases, and invalid inputs. Overall, the function performs well for basic format conversion but does not validate whether the date itself is valid (e.g., February 30 would still be accepted).