LAB ASSIGNMENT-8.3

Name: Sindhu Katakam

Enrollment number: 2303A52481

Batch-41

Task 1: Email Validation using TDD

Prompt:

generate general test cases for email input validation . it must contain a single @ and . characters ,must not start or end with special characters.cover all valid and invalid cases.

Code:

```
class EmailValidator:
    def __init__(self, email):
        self.email = email

    def is_valid_email(self):
        # Check if there is exactly one '@' character
        if self.email.count('@') != 1:
            return False

        # Split the email into local and domain parts
        local_part, domain_part = self.email.split('@')

        # Check if the local part and domain part are not empty
        if not local_part or not domain_part:
            return False

        # Check if the email starts or ends with special characters
        if self.email[0] in ['@', '.'] or self.email[-1] in ['@', '.']:
            return False

        # Check if there is at least one '.' character in the domain part
        if '.' not in domain_part:
            return False

        return True
```

```
test_cases = [
    "test@example.com",
    "user.name@domain.co",
    "user_name@domain.com",
    "username@sub.domain.com",
    "username@domain.co.in",
    "user-name@domain.com",
    "user.name+tag+sorting@example.com",
    "user.name@example.co.uk",
    "user.name@domain",
    "user.name@.com",
    "@example.com",
    "username@domain..com",
    "username@domain.c",
    "username@@domain.corporate"
]
def run_tests():
    for email in test_cases:
        validator = EmailValidator(email)
        result = validator.is_valid_email()
        print(f"Email: {email}, Valid: {result}")
if __name__ == "__main__":
    run_tests()
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/Local/
T_CODING/#assignment-8.3.py
Email: test@example.com, Valid: True
Email: user.name@domain.co, Valid: True
Email: user_name@domain.com, Valid: True
Email: username@sub.domain.com, Valid: True
Email: username@domain.co.in, Valid: True
Email: user-name@domain.com, Valid: True
Email: user.name+tag+sorting@example.com, Valid: True
Email: user.name@example.co.uk, Valid: True
Email: user.name@domain, Valid: False
```

Explanation:

These test cases help check whether the email validation works correctly in different situations. Valid cases confirm that properly formatted emails are accepted.Invalid cases check common mistakes like missing @, missing dots, or wrong positions of special characters.Edge cases such as empty strings and multiple @ symbols ensure the program handles errors safely.Testing both valid and invalid inputs increases confidence that the validator behaves correctly in real use.

Task 2: Grade Assignment using Loops

Prompt:
Write a Python function called assign_grade(score) that returns a grade based on the score:
90 to 100 gets "A"
80 to 89 gets "B"
70 to 79 gets "C"
60 to 69 gets "D"
Below 60 gets "F"
Test your function with these scores: 60, 70, 80, 90, -5, 105, and "eighty".
Make sure your function handles invalid inputs and boundary values correctly.
Code:

```
def assign_grade(score):
    if not isinstance(score, (int, float)):
        raise ValueError("Invalid input: score must be a number.")
    if score < 0 or score > 100:
        raise ValueError("Invalid input: score must be between 0 and 100.")

    if 90 <= score <= 100:
        return "A"
    elif 80 <= score < 90:
        return "B"
    elif 70 <= score < 80:
        return "C"
    elif 60 <= score < 70:
        return "D"
    else:
        return "F"
# Test cases
test_scores = [60, 70, 80, 90, -5, 105, "eighty"]
for score in test_scores:
    try:
        grade = assign_grade(score)
        print(f"Score: {score}, Grade: {grade}")
    except ValueError as e:
        print(f" {e}")
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/L
T_CODING/#assignment-8.3.py
Score: 60, Grade: D
Score: 70, Grade: C
Score: 80, Grade: B
Score: 90, Grade: A
 Invalid input: score must be between 0 and 100.
 Invalid input: score must be between 0 and 100.
 Invalid input: score must be a number.
PS D:\AI_ASSIT_CODING>
```

Explanation:

The function assign_grade(score) is designed to determine the letter grade based on a numerical score. It first checks if the input is a valid number and falls within the acceptable range (0-100). If the input is invalid, it raises an error with a clear message. If the input is valid, it uses conditional statements to assign the appropriate grade based on the defined score ranges. This ensures that the function handles edge cases and invalid inputs gracefully while providing accurate results for valid scores.

Task 3: Sentence Palindrome Checker

Prompt:

Write a Python program to check if a sentence is a palindrome or not.

Ignore capital/small letters, spaces, and punctuation marks.

Test your code with sentences that are palindromes and some that are not.

For example, "A man a plan a canal Panama" should return True.

Make sure your code works for all cases and ignores spaces and punctuation

Code:

```python
import string
def is_palindrome(sentence):
    # Remove spaces and punctuation, and convert to lowercase
    cleaned_sentence = ''.join(char for char in sentence if char.isalnum()).lower()

    # Check if the cleaned sentence is equal to its reverse
    return cleaned_sentence == cleaned_sentence[::-1]
# Test cases
test_sentences = [
    "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"
]
for sentence in test_sentences:
    result = is_palindrome(sentence)
    print(f" '{sentence}', Is palindrome: {result}")
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/Local/
T_CODING/#assignment-8.3.py
 'A man a plan a canal Panama', Is palindrome: True
 'No 'x' in Nixon', Is palindrome: True
 'Was it a car or a cat I saw?', Is palindrome: True
 'Not a palindrome', Is palindrome: False
 '12321', Is palindrome: True
 '12345', Is palindrome: False
```

Explanation:

The function is_palindrome(sentence) checks if a given sentence is a palindrome by first cleaning the input. It removes all spaces and punctuation marks, and converts the string to lowercase to ensure that the palindrome check is case-insensitive and ignores non-alphanumeric characters. Then, it compares the cleaned sentence with its reverse to determine if it is a palindrome. This approach allows the function to accurately identify palindromic sentences while ignoring irrelevant characters, making it robust for various inputs.

Task 4: ShoppingCart Class

Prompt:

Generate a shoppping cart program . the class has to include methods like add_item(name,price),remove_item(name),total_cost(). generate test cases for the shopping cart class.validate correct addition,removal and cost calculation. handle empty cart scenarios.calculate costs accurately and add and remove items correctly.

Code:

```
class ShoppingCart:
    def __init__(self):
        self.cart = {}

    def add_item(self, name, price):
        if not isinstance(name, str) or not isinstance(price, (int, float)):
            raise ValueError("Invalid input: name must be a string and price must be a number.")
        if price < 0:
            raise ValueError("Invalid input: price cannot be negative.")
        self.cart[name] = price

    def remove_item(self, name):
        if name in self.cart:
            del self.cart[name]
        else:
            raise ValueError("Item not found in cart.")

    def total_cost(self):
        return sum(self.cart.values())
# give all the test cases in single try catch block
try:
    cart = ShoppingCart()
    cart.add_item("Apple", 1.5)
    cart.add_item("Banana", 0.75)
    cart.add_item("Orange", 1.25)
```

```
    print(f"Total cost after adding items: {cart.total_cost()}")  # Expected: 3.5

    cart.remove_item("Banana")
    print(f"Total cost after removing Banana: {cart.total_cost()}")  # Expected: 2.75

    cart.remove_item("Grapes")  # This should raise an error
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/Local/
Total cost after adding items: 3.5
Total cost after removing Banana: 2.75
Error: Item not found in cart.
```

Explanation:

The ShoppingCart class allows users to manage their shopping cart by adding and removing items, as well as calculating the total cost. The add_item method checks for valid input and updates the cart accordingly, while the remove_item method ensures that only existing items can be removed. The total_cost method calculates the sum of all item prices in the cart. The test cases validate that items are added and removed correctly, and that the total cost is calculated accurately, while also handling scenarios where an attempt is made to remove an item that does not exist in the cart.

Task 5: Date Format Conversion

Prompt:

Write a Python function called convert_date_format(date_str) that takes a date in the format "YYYY-MM-DD" and returns it in the format "DD-MM-YYYY".

For example, if the input is "2023-10-15", the output should be "15-10-2023".

Test the function with different valid dates to make sure it works correctly. Also, handle invalid date formats gracefully by raising an appropriate error.

Code:

```python
def convert_date_format(date_str):
    try:
        year, month, day = date_str.split('-')
        if len(year) != 4 or len(month) != 2 or len(day) != 2:
            raise ValueError("Invalid date format: expected 'YYYY-MM-DD'.")
        return f"{day}-{month}-{year}"
    except ValueError as e:
        raise ValueError(f"Invalid input: {e}")
# Test cases
test_dates = [
    "2023-10-15",
    "1990-01-01",
    "2020-12-31",
    "2023-2-5",  # Invalid format
    "2023/10/15",  # Invalid format
    "15-10-2023"  # Invalid format
]
for date in test_dates:
    try:
        converted_date = convert_date_format(date)
        print(f"Original: {date}, Converted: {converted_date}")
    except ValueError as e:
        print(f"Error: {e}")
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/Local/Programs/Python/
T_CODING/#assignment-8.3.py
Original: 2023-10-15, Converted: 15-10-2023
Original: 1990-01-01, Converted: 01-01-1990
Original: 2020-12-31, Converted: 31-12-2020
Error: Invalid input: Invalid date format: expected 'YYYY-MM-DD'.
Error: Invalid input: not enough values to unpack (expected 3, got 1)
Error: Invalid input: Invalid date format: expected 'YYYY-MM-DD'.
PS D:\AI_ASSIT_CODING>
```

Explanation:

The function convert_date_format(date_str) is designed to take a date string in the format. "YYYY-MM-DD" and convert it to "DD-MM-YYYY". It first attempts to split the input string into year, month, and day components. It then checks if the components are in the correct. If the input is valid, it rearranges the components and returns the new date format. If the input is invalid, it raises a ValueError with a clear message indicating the issue.