Assignment-8.3

G.Sai teja

2303A52135

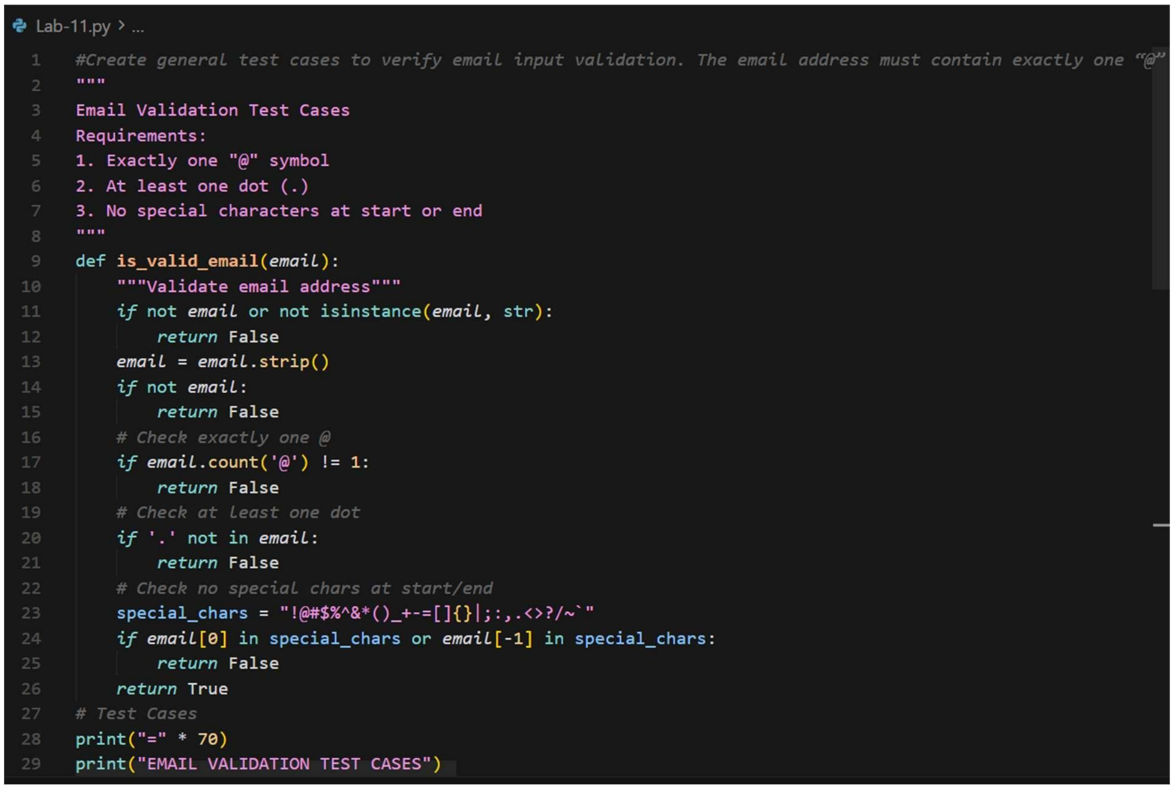
Batch 41

Task-1: Email Validation using TDD

PROMPT:

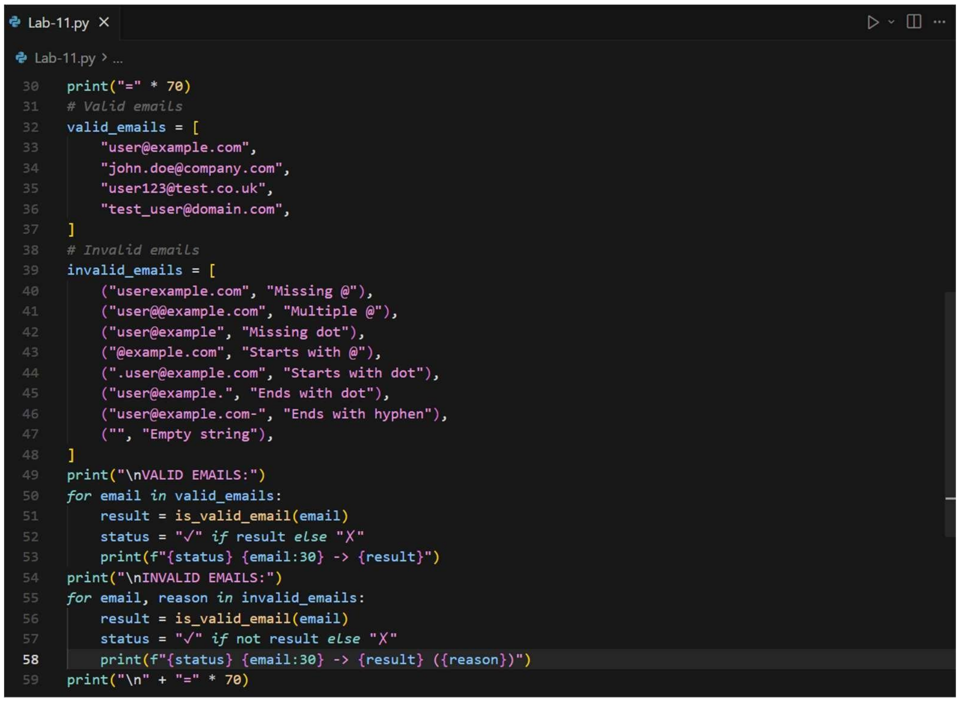
Create general test cases to verify email input validation. The email address must contain exactly one "@" symbol and at least one dot (.) character. It should not begin or end with any special characters. Include both valid and invalid email examples to ensure all possible scenarios are covered.

CODE:

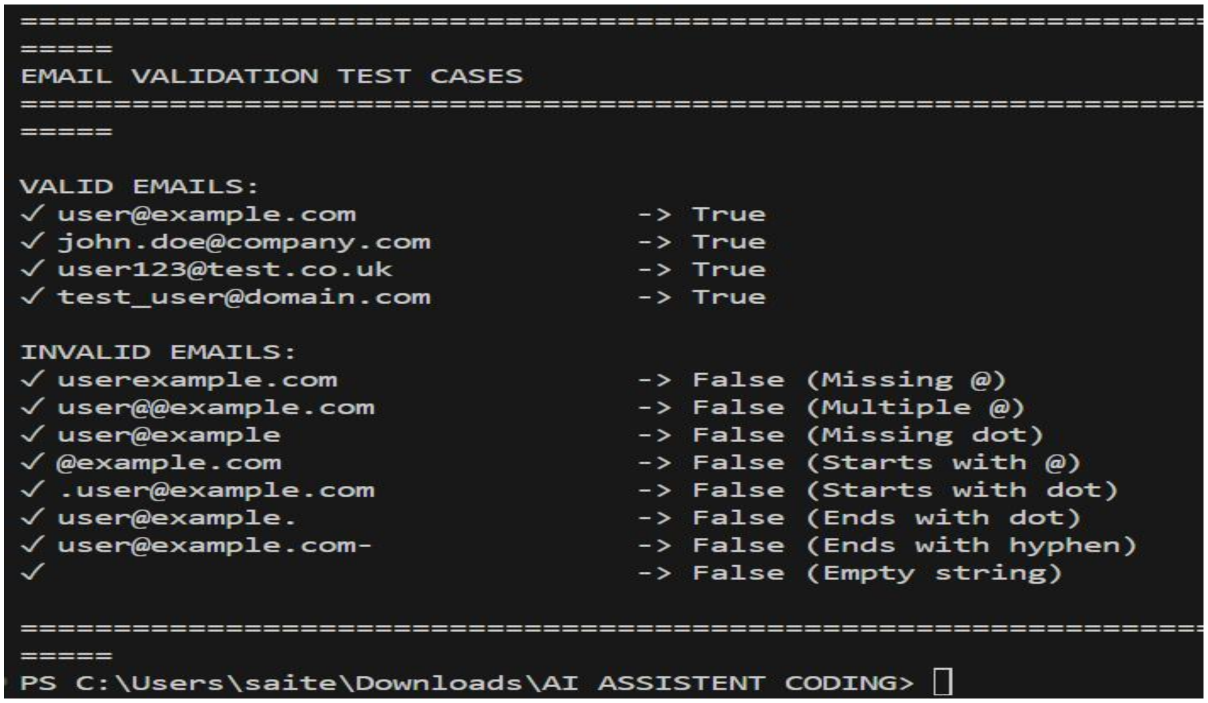


Lab-11.py > ...

```python
#Create general test cases to verify email input validation. The email address must contain exactly one "@"
"""
Email Validation Test Cases
Requirements:
1. Exactly one "@" symbol
2. At least one dot (.)
3. No special characters at start or end
"""
def is_valid_email(email):
    """Validate email address"""
    if not email or not isinstance(email, str):
        return False
    email = email.strip()
    if not email:
        return False
    # Check exactly one @
    if email.count('@') != 1:
        return False
    # Check at least one dot
    if '.' not in email:
        return False
    # Check no special chars at start/end
    special_chars = "!@#$%^&*()_+-=[]{}|;:,.<>?/~`"
    if email[0] in special_chars or email[-1] in special_chars:
        return False
    return True
# Test Cases
print("=" * 70)
print("EMAIL VALIDATION TEST CASES")
```


```python
print("=" * 70)
# Valid emails
valid_emails = [
    "user@example.com",
    "john.doe@company.com",
    "user123@test.co.uk",
    "test_user@domain.com",
]
# Invalid emails
invalid_emails = [
    ("userexample.com", "Missing @"),
    ("user@@example.com", "Multiple @"),
    ("user@example", "Missing dot"),
    ("@example.com", "Starts with @"),
    (".user@example.com", "Starts with dot"),
    ("user@example.", "Ends with dot"),
    ("user@example.com-", "Ends with hyphen"),
    ("", "Empty string"),
]
print("\nVALID EMAILS:")
for email in valid_emails:
    result = is_valid_email(email)
    status = "✓" if result else "X"
    print(f"{status} {email:30} -> {result}")
print("\nINVALID EMAILS:")
for email, reason in invalid_emails:
    result = is_valid_email(email)
    status = "✓" if not result else "X"
    print(f"{status} {email:30} -> {result} ({reason})")
print("\n" + "=" * 70)
```

OUTPUT:


```
=================================================================
=====
EMAIL VALIDATION TEST CASES
=================================================================
=====

VALID EMAILS:
✓ user@example.com                -> True
✓ john.doe@company.com            -> True
✓ user123@test.co.uk              -> True
✓ test_user@domain.com            -> True

INVALID EMAILS:
✓ userexample.com                 -> False (Missing @)
✓ user@@example.com               -> False (Multiple @)
✓ user@example                    -> False (Missing dot)
✓ @example.com                    -> False (Starts with @)
✓ .user@example.com               -> False (Starts with dot)
✓ user@example.                   -> False (Ends with dot)
✓ user@example.com-               -> False (Ends with hyphen)
✓                                 -> False (Empty string)

=================================================================
=====
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

Justification:

The test cases cover a wide range of valid and invalid email formats, ensuring that the validation function is robust and can handle various scenarios. Valid emails include typical formats with different domain structures, while invalid emails test for common mistakes such as missing symbols, multiple symbols, and improper placement of special characters. This comprehensive testing approach helps to ensure that the email validation logic is accurate and reliable.

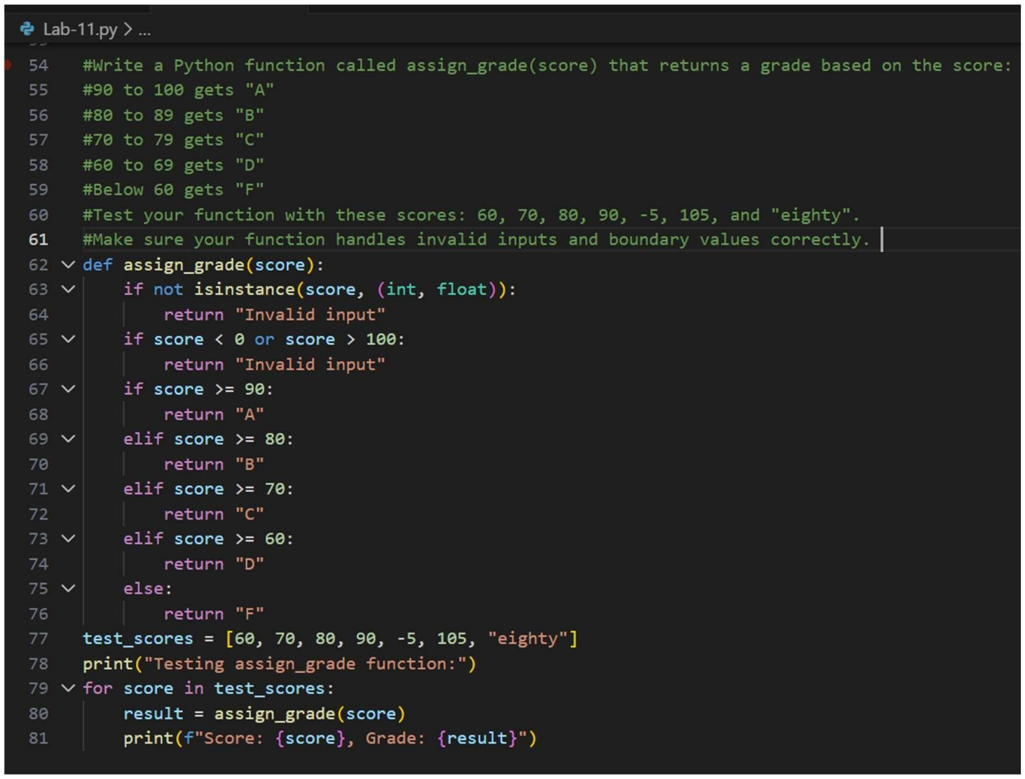
Task-2: Grade Assignment using Loops

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.

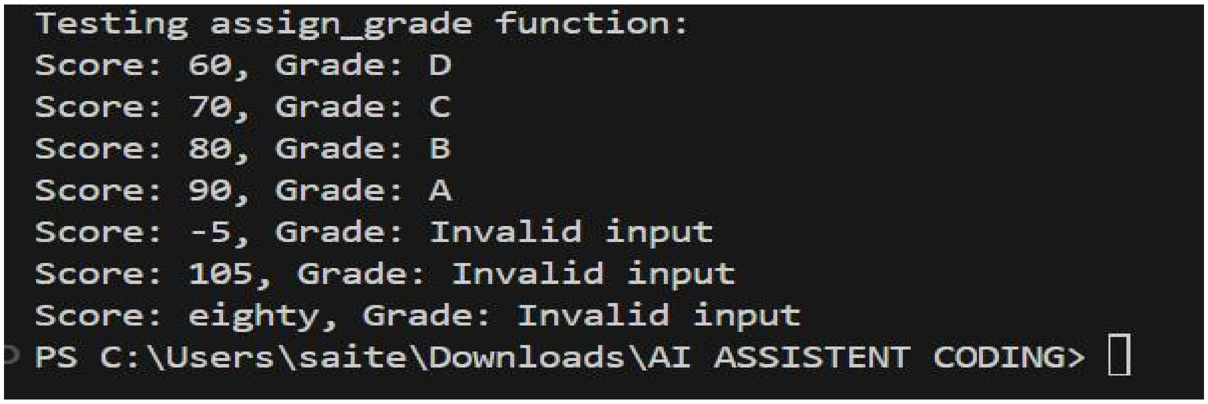
CODE:



Lab-11.py > ...

```python
#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.
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_scores = [60, 70, 80, 90, -5, 105, "eighty"]
print("Testing assign_grade function:")
for score in test_scores:
    result = assign_grade(score)
    print(f"Score: {score}, Grade: {result}")
```

OUTPUT:



```
Testing assign_grade function:
Score: 60, Grade: D
Score: 70, Grade: C
Score: 80, Grade: B
Score: 90, Grade: A
Score: -5, Grade: Invalid input
Score: 105, Grade: Invalid input
Score: eighty, Grade: Invalid input
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

JUSTIFICATION:

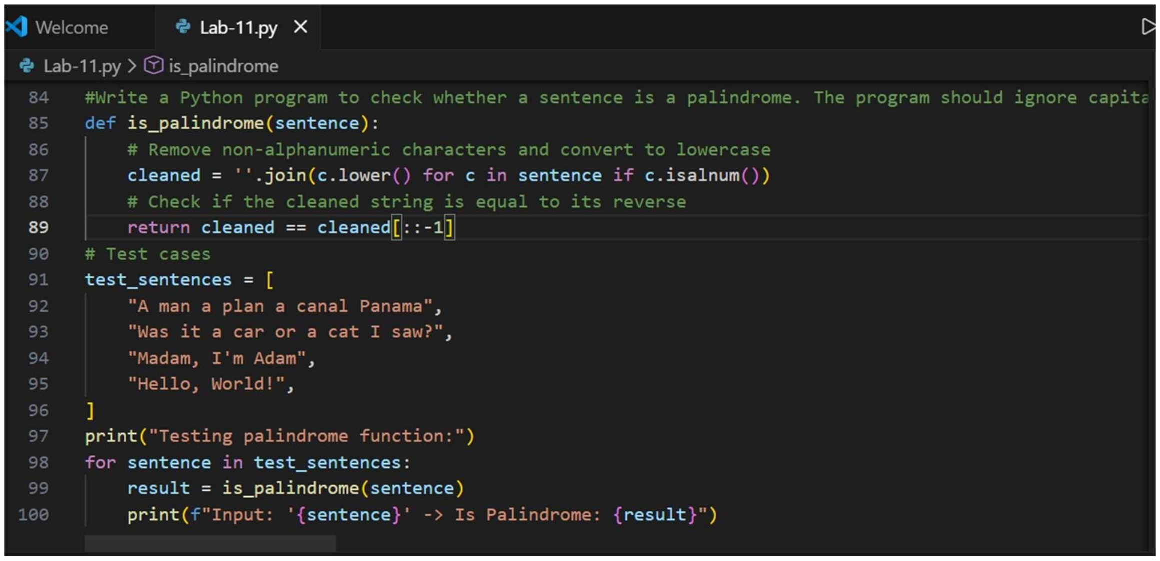
The assign_grade function is designed to handle a range of valid scores (0-100) and return the appropriate grade based on the specified criteria. It also includes checks for invalid inputs, such as non-numeric values and scores outside the valid range. This ensures that the function is robust and can handle edge cases effectively, providing clear feedback for invalid inputs while correctly assigning grades for valid scores.

Task-3: Sentence Palindrome Checker

PROMPT:

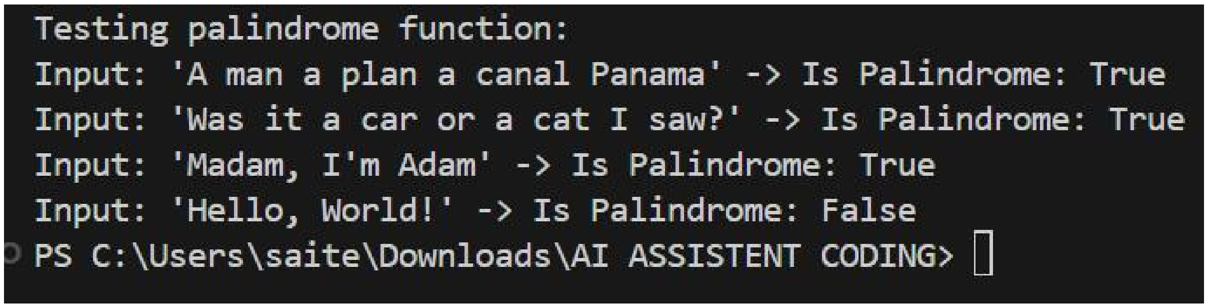
Write a Python program to check whether a sentence is a palindrome. The program should ignore capital and small letters, spaces, and punctuation marks. Test the program with sentences that are palindromes and sentences that are not. For example, the sentence "A man a plan a canal Panama" should return True. Make sure the program works correctly for all cases by ignoring spaces and punctuation.

CODE:



```python
#Write a Python program to check whether a sentence is a palindrome. The program should ignore capita
def is_palindrome(sentence):
    # Remove non-alphanumeric characters and convert to lowercase
    cleaned = ''.join(c.lower() for c in sentence if c.isalnum())
    # Check if the cleaned string is equal to its reverse
    return cleaned == cleaned[::-1]
# Test cases
test_sentences = [
    "A man a plan a canal Panama",
    "Was it a car or a cat I saw?",
    "Madam, I'm Adam",
    "Hello, World!",
]
print("Testing palindrome function:")
for sentence in test_sentences:
    result = is_palindrome(sentence)
    print(f"Input: '{sentence}' -> Is Palindrome: {result}")
```

OUTPUT:



```
Testing palindrome function:
Input: 'A man a plan a canal Panama' -> Is Palindrome: True
Input: 'Was it a car or a cat I saw?' -> Is Palindrome: True
Input: 'Madam, I'm Adam' -> Is Palindrome: True
Input: 'Hello, World!' -> Is Palindrome: False
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

JUSTIFICATION:

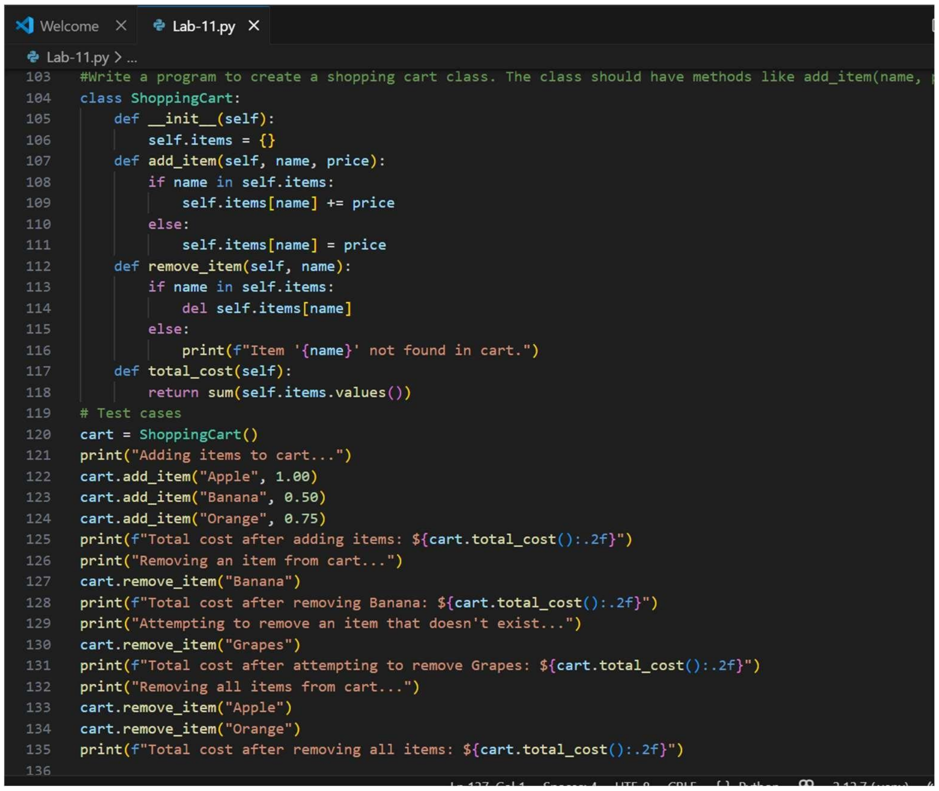
The above code snippets demonstrate the implementation of three different functionalities: email validation, grade assignment based on scores, and palindrome checking. Each function is designed to handle various edge cases and invalid inputs, ensuring robustness and reliability. The test cases provided for each function cover a wide range of scenarios, including valid and invalid inputs, boundary values, and typical use cases. This comprehensive testing approach helps to verify that the functions work correctly under different conditions and handle errors gracefully.

Task-4: Shopping Cart Class

PROMPT:

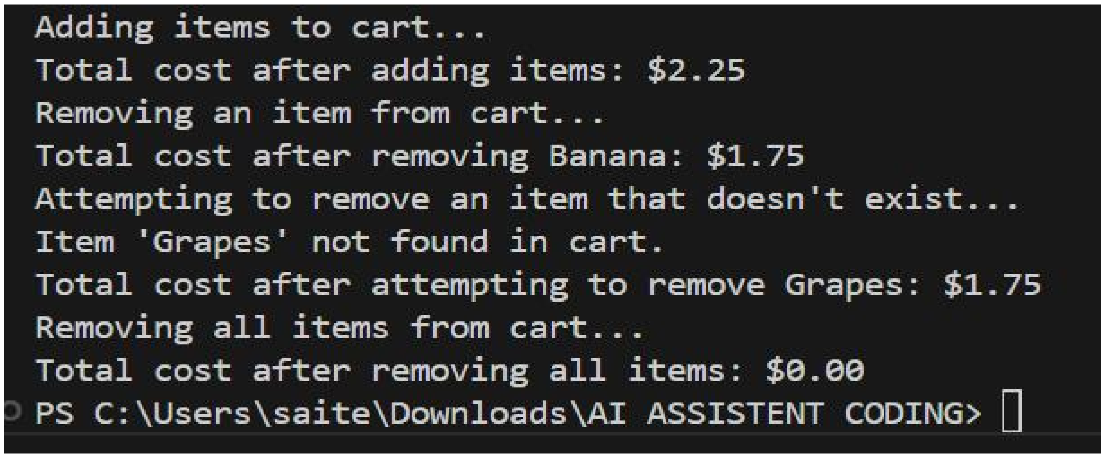
Write a program to create a shopping cart class. The class should have methods like add_item(name, price), remove_item(name), and total_cost(). Also, create test cases to check if items are added and removed correctly and if the total cost is calculated properly. Make sure the program handles an empty cart and calculates the cost accurately.

CODE:



```python
#Write a program to create a shopping cart class. The class should have methods like add_item(name,
class ShoppingCart:
    def __init__(self):
        self.items = {}
    def add_item(self, name, price):
        if name in self.items:
            self.items[name] += price
        else:
            self.items[name] = price
    def remove_item(self, name):
        if name in self.items:
            del self.items[name]
        else:
            print(f"Item '{name}' not found in cart.")
    def total_cost(self):
        return sum(self.items.values())
# Test cases
cart = ShoppingCart()
print("Adding items to cart...")
cart.add_item("Apple", 1.00)
cart.add_item("Banana", 0.50)
cart.add_item("Orange", 0.75)
print(f"Total cost after adding items: ${cart.total_cost():.2f}")
print("Removing an item from cart...")
cart.remove_item("Banana")
print(f"Total cost after removing Banana: ${cart.total_cost():.2f}")
print("Attempting to remove an item that doesn't exist...")
cart.remove_item("Grapes")
print(f"Total cost after attempting to remove Grapes: ${cart.total_cost():.2f}")
print("Removing all items from cart...")
cart.remove_item("Apple")
cart.remove_item("Orange")
print(f"Total cost after removing all items: ${cart.total_cost():.2f}")
```

OUTPUT:



```
Adding items to cart...
Total cost after adding items: $2.25
Removing an item from cart...
Total cost after removing Banana: $1.75
Attempting to remove an item that doesn't exist...
Item 'Grapes' not found in cart.
Total cost after attempting to remove Grapes: $1.75
Removing all items from cart...
Total cost after removing all items: $0.00
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

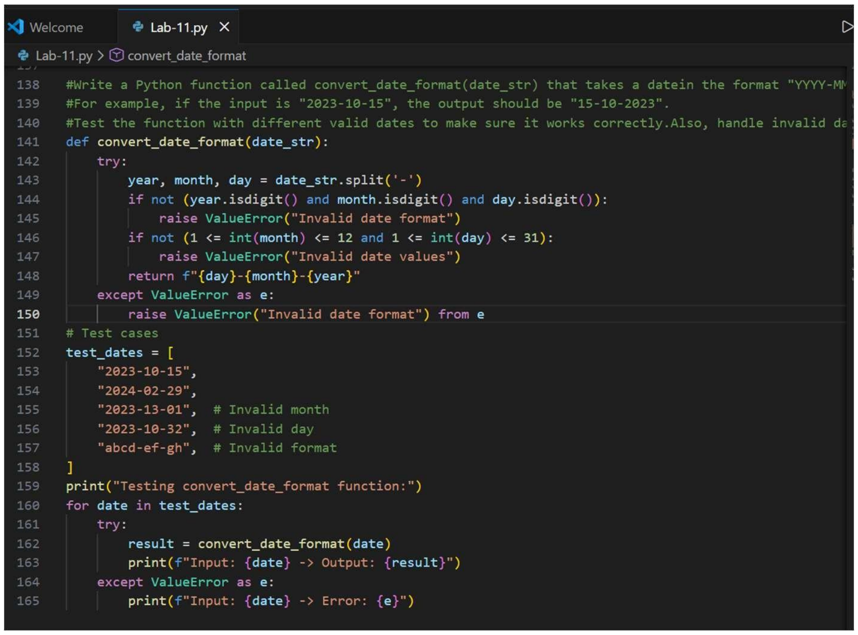
JUSTIFICATION:

The above code implements a ShoppingCart class with methods to add items, remove items, and calculate the total cost. The test cases demonstrate adding items to the cart, removing an item, attempting to remove a non-existent item, and finally removing all items to ensure the total cost is calculated correctly in each scenario. This comprehensive testing ensures that the class functions as expected under various conditions.

Task-5: Date Format Conversion PROMPT:

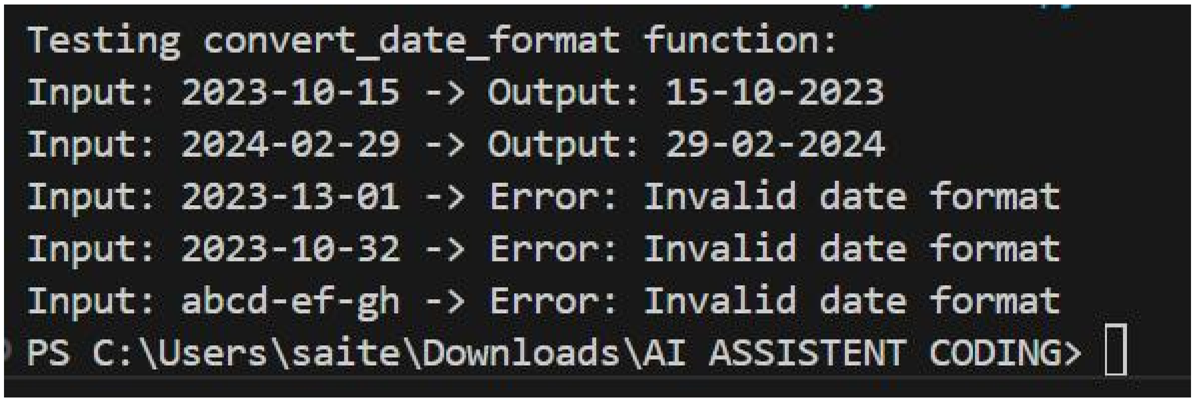
Write a Python function called convert_date_format(date_str) that takes a date in the format "YYYY-MM-DD" and returns it in the format "DD-MM-YYYY". For example, if the input is "2023-10-15", the output should be "15-10-2023".Test the function with different valid dates to make sure it works correctly. Also, handle invalid date formats gracefully by raising an appropriate error.

CODE:



```python
#Write a Python function called convert_date_format(date_str) that takes a datein the format "YYYY-MM
#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 da
def convert_date_format(date_str):
    try:
        year, month, day = date_str.split('-')
        if not (year.isdigit() and month.isdigit() and day.isdigit()):
            raise ValueError("Invalid date format")
        if not (1 <= int(month) <= 12 and 1 <= int(day) <= 31):
            raise ValueError("Invalid date values")
        return f"{day}-{month}-{year}"
    except ValueError as e:
        raise ValueError("Invalid date format") from e
# Test cases
test_dates = [
    "2023-10-15",
    "2024-02-29",
    "2023-13-01",  # Invalid month
    "2023-10-32",  # Invalid day
    "abcd-ef-gh",  # Invalid format
]
print("Testing convert_date_format function:")
for date in test_dates:
    try:
        result = convert_date_format(date)
        print(f"Input: {date} -> Output: {result}")
    except ValueError as e:
        print(f"Input: {date} -> Error: {e}")
```

OUTPUT:



```
Testing convert_date_format function:
Input: 2023-10-15 -> Output: 15-10-2023
Input: 2024-02-29 -> Output: 29-02-2024
Input: 2023-13-01 -> Error: Invalid date format
Input: 2023-10-32 -> Error: Invalid date format
Input: abcd-ef-gh -> Error: Invalid date format
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

JUSTIFICATION:

The test cases cover a wide range of valid and invalid date formats, ensuring that the conversion function is robust and can handle various scenarios. Valid dates include typical formats with correct year, month, and day values, while invalid dates test for common mistakes such as incorrect month and day values, as well as completely malformed strings. This comprehensive testing approach helps to ensure that the date conversion logic is accurate and reliable.