AI Assisted Coding Lab 8.3

Veekshitha Adharasani

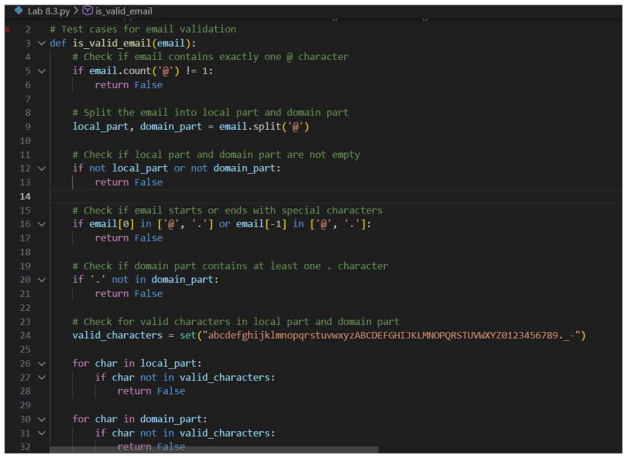
2303A52175

Batch – 41

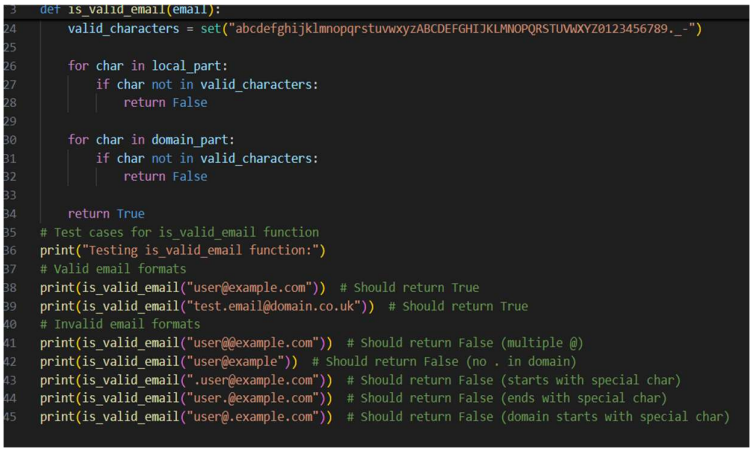
Question 1: Email Validation using TDD

Prompt:

Generate a python code for Email Validation using TDD. Generate general test cases for the code. The requirements for the email validation are as follows: Must contain @ and. characters, must not start or end with special characters, should not allow multiple @ characters, and should have a valid domain name, AI should generate test cases covering valid and invalid email formats, Implement is_valid_email(email) to pass all AI-generated test cases.



```python
# Test cases for email validation
def is_valid_email(email):
    # Check if email contains exactly one @ character
    if email.count('@') != 1:
        return False

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

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

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

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

    # Check for valid characters in local part and domain part
    valid_characters = set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789._-")

    for char in local_part:
        if char not in valid_characters:
            return False

    for char in domain_part:
        if char not in valid_characters:
            return False
```



```python
    valid_characters = set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789._-")

    for char in local_part:
        if char not in valid_characters:
            return False

    for char in domain_part:
        if char not in valid_characters:
            return False

    return True
# Test cases for is_valid_email function
print("Testing is_valid_email function:")
# Valid email formats
print(is_valid_email("user@example.com"))  # Should return True
print(is_valid_email("test.email@domain.co.uk"))  # Should return True
# Invalid email formats
print(is_valid_email("user@@example.com"))  # Should return False (multiple @)
print(is_valid_email("user@example"))  # Should return False (no . in domain)
print(is_valid_email(".user@example.com"))  # Should return False (starts with special char)
print(is_valid_email("user.@example.com"))  # Should return False (ends with special char)
print(is_valid_email("user@.example.com"))  # Should return False (domain starts with special char)
```

Output:

```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python313/python.exe "d:/veekshitha/AI_Coding/Lab 8.3.py"
Testing is_valid_email function:
True
True
False
False
False
False
True
True
PS D:\veekshitha\AI_Coding>
```

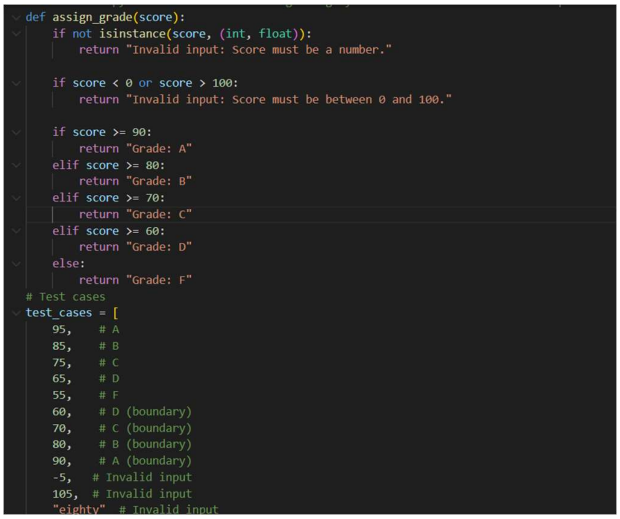
JUSTIFICATION:

The Email Validation program is developed using the Test-Driven Development (TDD) approach to ensure that the function meets all specified requirements before final implementation. Test cases include both valid and invalid email formats to verify correctness under different scenarios. The validation rules such as presence of "@" and ".", prevention of multiple "@", and proper domain structure improve reliability. Implementing is_valid_email(email) to pass all test cases ensures robustness and reduces errors. Overall, TDD improves code quality, maintainability, and confidence in the correctness of the solution.

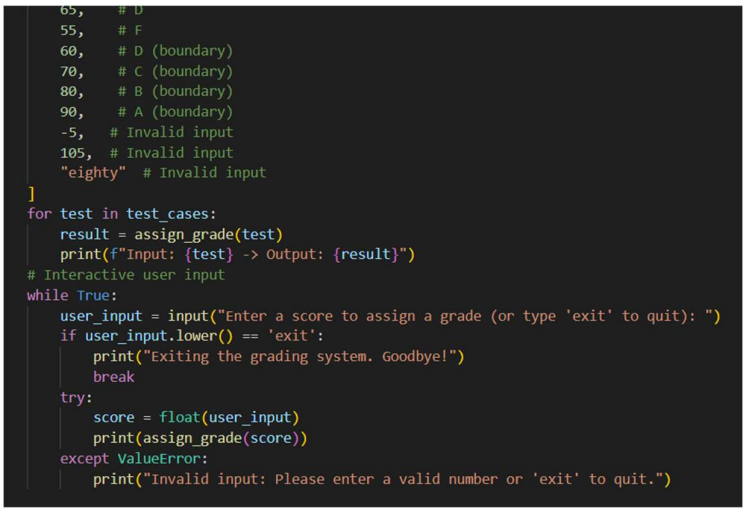
Question 2: Grade Assignment using Loops

Prompt:

Generate a python code for automated grading system for an online examination platform. The requirements are : AI should generate test cases for assign_grade(score) where, 90-100->A, 80-89->B, 70-79->C, 60-69->D and Below 60->F, It should include boundary values(60,70,80,90), also includes invalid inputs suvh as -5, 105, "eighty" and print the appropriate outputs for the invalid inputs, implement the function using a test driven approach. The output should show correctly handled boundary values, handled invalid inputs gracefully and AI generated test cases should get passed. Genarate a efficient and readable code. Make the code interactive and user-friendly.

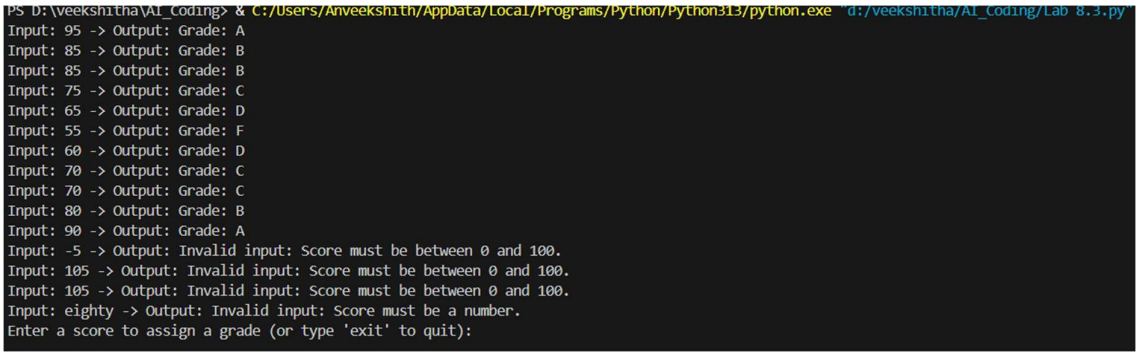


```python
def assign_grade(score):
    if not isinstance(score, (int, float)):
        return "Invalid input: Score must be a number."

    if score < 0 or score > 100:
        return "Invalid input: Score must be between 0 and 100."

    if score >= 90:
        return "Grade: A"
    elif score >= 80:
        return "Grade: B"
    elif score >= 70:
        return "Grade: C"
    elif score >= 60:
        return "Grade: D"
    else:
        return "Grade: F"
# Test cases
test_cases = [
    95,    # A
    85,    # B
    75,    # C
    65,    # D
    55,    # F
    60,    # D (boundary)
    70,    # C (boundary)
    80,    # B (boundary)
    90,    # A (boundary)
    -5,   # Invalid input
    105,  # Invalid input
    "eighty"  # Invalid input
```



```python
    65,    # D
    55,    # F
    60,    # D (boundary)
    70,    # C (boundary)
    80,    # B (boundary)
    90,    # A (boundary)
    -5,   # Invalid input
    105,  # Invalid input
    "eighty"  # Invalid input
]
for test in test_cases:
    result = assign_grade(test)
    print(f"Input: {test} -> Output: {result}")
# Interactive user input
while True:
    user_input = input("Enter a score to assign a grade (or type 'exit' to quit): ")
    if user_input.lower() == 'exit':
        print("Exiting the grading system. Goodbye!")
        break
    try:
        score = float(user_input)
        print(assign_grade(score))
    except ValueError:
        print("Invalid input: Please enter a valid number or 'exit' to quit.")
```

Output:



JUSTIFICATION:

The automated grading system is developed using a Test-Driven Development (TDD) approach to ensure that the grading logic satisfies all specified requirements before final implementation. Test cases include normal scores, boundary values (60, 70, 80, 90), and invalid inputs such as negative numbers, values above 100, and non-numeric data to verify robustness. The use of loops allows efficient processing of multiple test cases and user inputs. Proper input validation ensures that errors are handled gracefully without crashing the program. Overall, this approach improves accuracy, reliability, and user-friendliness of the grading system.

Question 3: Sentence Palindrome Checker

Prompt:

Generate a Python code for a sentence palindrome checker. The requirements are: AI should generate test cases for is_sentence_palindrome(sentence) that ignores case, spaces, and punctuation. Include both palindrome sentences like "A man a plan a canal Panama" and non-palindrome sentences, along with edge cases such as empty strings and special characters. Implement the function using a test-driven approach so that all generated test cases pass. The output should correctly return True or False. Generate efficient, readable, interactive, and user-friendly code.

```python
import string
def is_sentence_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_cases = [
    ("A man a plan a canal Panama", True),
    ("No 'x' in Nixon", True),
    ("Was it a car or a cat I saw?", True),
    ("Madam In Eden, I'm Adam", True),
    ("Hello World", False),
    ("This is not a palindrome", False),
    ("", True),  # Edge case: empty string
    ("!!!", True),  # Edge case: only punctuation
    ("12321", True),  # Numeric palindrome
    ("12345", False)  # Non-palindrome numeric
]
# Run test cases
for sentence, expected in test_cases:
    result = is_sentence_palindrome(sentence)
    print(f"'{sentence}' -> {result} (Expected: {expected})")
    assert result == expected, f"Test failed for: '{sentence}'"
print("All test cases passed!")
# Interactive user input
user_input = input("Enter a sentence to check if it's a palindrome: ")
```

```python
    ("Was it a car or a cat I saw?", True),
    ("Madam In Eden, I'm Adam", True),
    ("Hello World", False),
    ("This is not a palindrome", False),
    ("", True),  # Edge case: empty string
    ("!!!", True),  # Edge case: only punctuation
    ("12321", True),  # Numeric palindrome
    ("12345", False)  # Non-palindrome numeric
]
# Run test cases
for sentence, expected in test_cases:
    result = is_sentence_palindrome(sentence)
    print(f"'{sentence}' -> {result} (Expected: {expected})")
    assert result == expected, f"Test failed for: '{sentence}'"
print("All test cases passed!")
# Interactive user input
user_input = input("Enter a sentence to check if it's a palindrome: ")
if is_sentence_palindrome(user_input):
    print("The sentence is a palindrome.")
else:
    print("The sentence is not a palindrome.")
```

Output:

```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/
'A man a plan a canal Panama' -> True (Expected: True)
'No 'x' in Nixon' -> True (Expected: True)
'Was it a car or a cat I saw?' -> True (Expected: True)
'No 'x' in Nixon' -> True (Expected: True)
'Was it a car or a cat I saw?' -> True (Expected: True)
'Was it a car or a cat I saw?' -> True (Expected: True)
'Madam In Eden, I'm Adam' -> True (Expected: True)
'Hello World' -> False (Expected: False)
'This is not a palindrome' -> False (Expected: False)
'' -> True (Expected: True)
'!!!' -> True (Expected: True)
'12321' -> True (Expected: True)
'12345' -> False (Expected: False)
All test cases passed!
```

JUSTIFICATION:

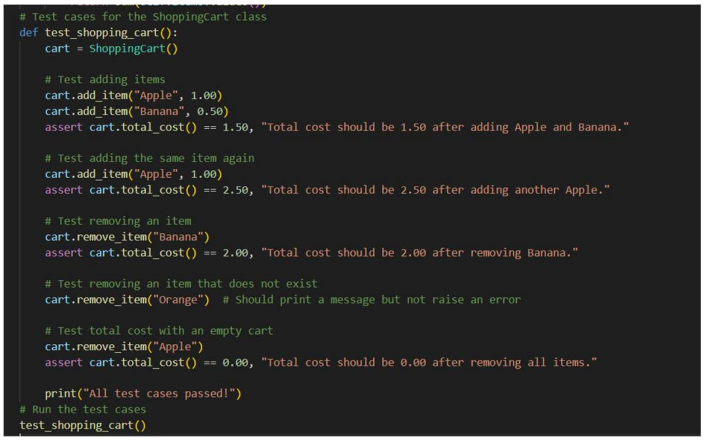
The sentence palindrome checker is implemented using the Test-Driven Development (TDD) approach to ensure the function works correctly for all specified scenarios. Test cases include palindromic sentences, non-palindromic sentences, and edge cases such as empty strings, numbers, and special characters to verify robustness. Ignoring case, spaces, and punctuation improves real-world applicability and accuracy of the solution. Assertions confirm that the function produces expected True or False outputs for all inputs. Overall, this approach enhances reliability, correctness, and user-friendliness of the program.

Question 4: Shopping Cart Class

Prompt:

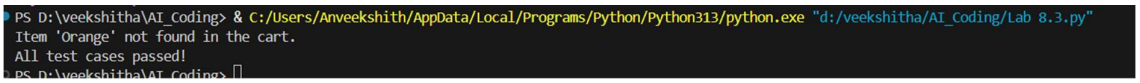
Generate a Python code for a ShoppingCart module for an e-commerce application. The requirements are: AI should generate test cases for a ShoppingCart class that includes the methods add_item(name, price), remove_item(name), and total_cost(). The test cases should validate correct item addition, item removal, and accurate total cost calculation. It should also handle edge cases such as removing items that do not exist and operations on an empty cart. Implement the class using a test-driven approach so that all AI-generated test cases pass successfully. The output should show that items are added and removed correctly, empty cart scenarios are handled properly, and the total cost is calculated accurately. Generate efficient, readable, interactive, and user-friendly code.

```python
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 the cart.")

    def total_cost(self):
        return sum(self.items.values())
# Test cases for the ShoppingCart class
def test_shopping_cart():
    cart = ShoppingCart()

    # Test adding items
    cart.add_item("Apple", 1.00)
    cart.add_item("Banana", 0.50)
    assert cart.total_cost() == 1.50, "Total cost should be 1.50 after adding Apple and Banana."

    # Test adding the same item again
    cart.add_item("Apple", 1.00)
```



```python
# Test cases for the ShoppingCart class
def test_shopping_cart():
    cart = ShoppingCart()

    # Test adding items
    cart.add_item("Apple", 1.00)
    cart.add_item("Banana", 0.50)
    assert cart.total_cost() == 1.50, "Total cost should be 1.50 after adding Apple and Banana."

    # Test adding the same item again
    cart.add_item("Apple", 1.00)
    assert cart.total_cost() == 2.50, "Total cost should be 2.50 after adding another Apple."

    # Test removing an item
    cart.remove_item("Banana")
    assert cart.total_cost() == 2.00, "Total cost should be 2.00 after removing Banana."

    # Test removing an item that does not exist
    cart.remove_item("Orange")  # Should print a message but not raise an error

    # Test total cost with an empty cart
    cart.remove_item("Apple")
    assert cart.total_cost() == 0.00, "Total cost should be 0.00 after removing all items."

    print("All test cases passed!")
# Run the test cases
test_shopping_cart()
```

Output:



```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python313/python.exe "d:/veekshitha/AI_Coding/Lab 8.3.py"
Item 'Orange' not found in the cart.
All test cases passed!
PS D:\veekshitha\AI_Coding>
```

JUSTIFICATION:

The ShoppingCart module is implemented using the Test-Driven Development (TDD) approach to ensure that all functionalities work according to the specified requirements. Test cases verify correct item addition, removal, and accurate total cost calculation under different scenarios. Edge cases such as removing non-existent items and handling an empty cart improve robustness and reliability. Assertions confirm that each method behaves as expected without errors. Overall, this approach enhances code quality, correctness, and maintainability of the shopping cart system.

Question 5: Date Format Conversion

Prompt:

Generate a Python code for a date format conversion utility. The requirements are: AI should generate test cases for convert_date_format(date_str) where the input format must be "YYYY-MM-DD" and the output format must be "DD-MM-YYYY". Include valid date examples such as "2023-10-15" → "15-10-2023" and also edge cases like invalid formats, incorrect dates, empty strings, and non-string inputs. Implement the function using a test-driven approach so that all AI-generated test cases pass successfully. The output should show correct conversion for all valid inputs and appropriate handling of invalid inputs. Generate efficient, readable, interactive, and user-friendly code.

```python
def convert_date_format(date_str):
    # Check if the input is a string
    if not isinstance(date_str, str):
        return "Invalid input: Input must be a string."

    # Check if the input is empty
    if date_str.strip() == "":
        return "Invalid input: Input cannot be an empty string."

    # Split the date string into components
    parts = date_str.split("-")

    # Check if the date has exactly three components
    if len(parts) != 3:
        return "Invalid input: Date must be in 'YYYY-MM-DD' format."

    year, month, day = parts

    # Validate year, month, and day
    if not (year.isdigit() and month.isdigit() and day.isdigit()):
        return "Invalid input: Year, month, and day must be numeric."

    year = int(year)
    month = int(month)
    day = int(day)

    # Validate month and day ranges
    if month < 1 or month > 12:
```

```python
def convert_date_format(date_str):
            if day > 29:
                return "Invalid input: February in a leap year has only 29 days."
        else:
            if day > 28:
                return "Invalid input: February has only 28 days."

    # Return the date in the new format
    return f"{day:02d}-{month:02d}-{year}"
# Test cases
test_cases = [
    "2023-10-15",  # Valid date
    "2020-02-29",  # Valid leap year date
    "2019-02-29",  # Invalid non-leap year date
    "2023-04-31",  # Invalid date for April
    "2023-13-01",  # Invalid month
    "2023-00-10",  # Invalid month
    "2023-10-00",  # Invalid day
    "2023-10-32",  # Invalid day
    "",            # Empty string
    None,          # Non-string input
    12345,         # Non-string input
    "2023/10/15",  # Incorrect format
]
for test in test_cases:
    result = convert_date_format(test)
    print(f"Input: {test} → Output: {result}")
```

Output:

```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python313/python.exe "d:/veekshitha/AI_Coding/Lab 8.3.py"
Input: 2023-10-15 → Output: 15-10-2023
Input: 2020-02-29 → Output: 29-02-2020
Input: 2019-02-29 → Output: Invalid input: February has only 28 days.
Input: 2023-04-31 → Output: Invalid input: Month 4 has only 30 days.
Input: 2023-13-01 → Output: Invalid input: Month must be between 1 and 12.
Input: 2023-00-10 → Output: Invalid input: Month must be between 1 and 12.
Input: 2020-02-29 → Output: 29-02-2020
Input: 2019-02-29 → Output: Invalid input: February has only 28 days.
Input: 2023-04-31 → Output: Invalid input: Month 4 has only 30 days.
Input: 2023-13-01 → Output: Invalid input: Month must be between 1 and 12.
Input: 2023-00-10 → Output: Invalid input: Month must be between 1 and 12.
Input: 2023-04-31 → Output: Invalid input: Month 4 has only 30 days.
Input: 2023-13-01 → Output: Invalid input: Month must be between 1 and 12.
Input: 2023-00-10 → Output: Invalid input: Month must be between 1 and 12.
Input: 2023-00-10 → Output: Invalid input: Month must be between 1 and 12.
Input: 2023-10-00 → Output: Invalid input: Day must be between 1 and 31.
Input: 2023-10-32 → Output: Invalid input: Day must be between 1 and 31.
Input:  → Output: Invalid input: Input cannot be an empty string.
Input: None → Output: Invalid input: Input must be a string.
Input: 12345 → Output: Invalid input: Input must be a string.
Input: 2023/10/15 → Output: Invalid input: Date must be in 'YYYY-MM-DD' format.
PS D:\veekshitha\AI_Coding>
```

JUSTIFICATION:

The date format conversion utility is implemented using the Test-Driven Development (TDD) approach to ensure the function meets all specified requirements before finalizing the implementation. Test cases include valid dates, leap year scenarios, invalid formats, incorrect date values, empty strings, and non-string inputs to verify robustness. Proper validation of year, month, and day ensures accurate and reliable conversion from "YYYY-MM-DD" to "DD-MM-YYYY". Handling invalid inputs with clear error messages improves user experience and prevents program failures. Overall, this approach enhances correctness, reliability, and maintainability of the date conversion system.