

AI Assisted Coding

Week-8.3

Ht no-2303A52169

Batch-45

Task 1: Email Validation using TDD

Scenario

You are developing a user registration system that requires reliable email input validation.

Requirements

- Must contain @ and . characters
- Must not start or end with special characters
- Should not allow multiple @ symbols
- AI should generate test cases covering valid and invalid email formats
- Implement is_valid_email(email) to pass all AI-generated test cases

Expected Output

- Python function for email validation
- All AI-generated test cases pass successfully
- Invalid email formats are correctly rejected
- Valid email formats return True

Prompt:

#Generate a python code code for developing the user registration with email input validation and it must contain the special characters like "@" and ". ". And it must not start or end with these special characters. Even it should not allow the multiple "@" symbols . Generate the test cases covering valid and invalid email formats. If the ouput has valid email formats return true if not correctly rejected all the generated testcase must pass sucessfully .

Code:

```
def is_valid_email(email: str) -> bool:  
    # Condition 1: Email must not start or end with '@' or '.'  
    if email.startswith('@') or email.startswith('.') or \  
        email.endswith('@') or email.endswith('.'):\  
        return False  
  
    # Condition 2: Must contain exactly one '@'  
    if email.count('@') != 1:  
        return False  
  
    # Condition 3: Must contain at least one '.'  
    if '.' not in email:  
        return False  
  
    # Split local and domain parts  
    local_part, domain_part = email.split('@')  
  
    # Condition 4: Local and domain parts must not be empty  
    if not local_part or not domain_part:  
        return False  
  
    # Condition 5: '.' must exist in domain part  
    if '.' not in domain_part:  
        return False  
  
    return True  
# Test cases  
print(is_valid_email("test@example.com")) # Should return True
```

```

print(is_valid_email("test@"))      # Should return False
print(is_valid_email("@example.com")) # Should return False
print(is_valid_email("test@example")) # Should return False
print(is_valid_email("test@example..com")) # Should return False
print(is_valid_email("test@.com"))     # Should return False
print(is_valid_email("test@example.com")) # Should return False
print(is_valid_email(""))            # Should return False
print(is_valid_email("test@@example.com")) # Should return False

```

Output:

```

PS C:\Users\vars\rsh/OneDrive/Des
True
False
False
False
True
True
True
False
False
False

```

Justification:

The function `is_valid_email` checks for the presence of exactly one '@' symbol and at least one '.' character, which are essential components of a valid email address. It also ensures that the email does not start or end with special characters, which is a common requirement for email validation. The test cases cover various valid and invalid email formats to ensure the function behaves as expected in different scenarios.

Task 2: Grade Assignment using Loops

Scenario

You are building an automated grading system for an online examination platform.

Requirements

- AI should generate test cases for `assign_grade(score)` where:
 - 90–100 → A
 - 80–89 → B
 - 70–79 → C
 - 60–69 → D
 - Below 60 → F
- Include boundary values (60, 70, 80, 90)
- Include invalid inputs such as -5, 105, "eighty"
- Implement the function using a test-driven approach

Expected Output

- Grade assignment function implemented in Python
- Boundary values handled correctly
- Invalid inputs handled gracefully
- All AI-generated test cases pass

Prompt:

#Generate the code for automated grading system for an online examination platform. The boundary values (60, 70, 80, 90). Using the function grade assignment, Include invalid inputs such as -5, 105, "eighty"

```
#Implement the function using a test-driven approach
Code:
def assign_grade(score: int) -> str:
    if score < 0 or score > 100:
        return "Invalid score"
    elif score < 60:
        return "D"
    elif score < 70:
        return "C"
    elif score < 80:
        return "B"
    elif score < 90:
        return "A"
    else:
        return "A"
# Test cases
print(assign_grade(59)) # Should return "F"
print(assign_grade(60)) # Should return "D"
print(assign_grade(69)) # Should return "D"
print(assign_grade(70)) # Should return "C"
print(assign_grade(79)) # Should return "C"
print(assign_grade(80)) # Should return "B"
print(assign_grade(89)) # Should return "B"
```

Output:

```
D
C
C
B
B
A
A
```

Justification:

The function `assign_grade` takes a score as input and first checks if the input is a number (either int or float). If not, it returns an error message. Then it checks if the score is within the valid range of 0 to 100. If it's outside this range, it returns another error message. Finally, it assigns a grade based on the score using conditional statements. The test cases cover valid scores for each grade category as well as invalid inputs to ensure the function behaves correctly in all scenarios.

Task 3: Sentence Palindrome Checker

Scenario

You are developing a text-processing utility to analyze sentences.

Requirements

- AI should generate test cases for `is_sentence_palindrome(sentence)`
- Ignore case, spaces, and punctuation
- Test both palindromic and non-palindromic sentences
- Example:

– "A man a plan a canal Panama" → True

Expected Output

- Function correctly identifies sentence palindromes
- Case and punctuation are ignored
- Returns True or False accurately
- All AI-generated test cases pass

Prompt:

#Generate a python code for text-processing utility to analyze sentence.,it should generate test cases for `is_sentence_palindrome(sentence)`,Ignore case, spaces, and punctuation,Test both palindromic and non-palindromic sentences

Code:

```
import string
def is_sentence_palindrome(sentence: str) -> bool:
    # 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 a palindrome
    return cleaned_sentence == cleaned_sentence[::-1]

# Test cases
print(is_sentence_palindrome("A man, a plan, a canal, Panama")) # Should return True
print(is_sentence_palindrome("No 'x' in Nixon")) # Should return True
print(is_sentence_palindrome("Hello, World!")) # Should return False
print(is_sentence_palindrome("Was it a car or a cat I saw?")) # Should return True
print(is_sentence_palindrome("This is not a palindrome")) # Should return False
print(is_sentence_palindrome("Madam In Eden, I'm Adam")) # Should return True
print(is_sentence_palindrome("12321")) # Should return True
print(is_sentence_palindrome("12345")) # Should return False
```

Output:

- PS C:\Users\varsh\OneDrive\Desktop\ai coding> & C:\top\ai coding\11.PY


```
True
True
False
True
False
True
True
False
```
- PS C:\Users\varsh\OneDrive\Desktop\ai coding> □

Justification:

The function `is_sentence_palindrome` takes a sentence as input and processes it to determine if it is a palindrome. It first removes all spaces and punctuation from the sentence and converts it 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. If they are the same, the function returns True, indicating that the sentence is a palindrome; otherwise, it returns False. The test cases include both palindromic and non-palindromic sentences to validate the function's correctness.

Task 4: ShoppingCart Class

Scenario

You are designing a basic shopping cart module for an e-commerce application.

Requirements

- AI should generate test cases for the ShoppingCart class
- Class must include the following methods:
 - add_item(name, price)
 - remove_item(name)
 - total_cost()
- Validate correct addition, removal, and cost calculation
- Handle empty cart scenarios

Expected Output

- Fully implemented ShoppingCart class
- All methods pass AI-generated test cases
- Total cost is calculated accurately
- Items are added and removed correctly

Prompt:

#Generate the code which include the class methods add_item(name, price) ,remove_item(name) and total_cost() and Validate correct addition, removal, and cost calculation in a shopping cart system. Implement the class and its methods, and include test cases to ensure functionality.

Code:

```
class ShoppingCart:  
    def __init__(self):  
        self.items = {}  
  
    def add_item(self, name: str, price: float):  
        if name in self.items:  
            self.items[name] += price  
        else:  
            self.items[name] = price  
  
    def remove_item(self, name: str):  
        if name in self.items:  
            del self.items[name]  
  
    def total_cost(self) -> float:  
        return sum(self.items.values())  
  
# Test cases  
cart = ShoppingCart()  
cart.add_item("Apple", 1.0)  
cart.add_item("Banana", 0.5)  
print(cart.total_cost()) # Should return 1.5  
cart.add_item("Apple", 1.0)  
print(cart.total_cost()) # Should return 2.5  
cart.remove_item("Banana")  
print(cart.total_cost()) # Should return 2.0  
cart.remove_item("Orange") # Removing non-existent item should not affect total cost  
print(cart.total_cost()) # Should return 2.0  
cart.remove_item("Apple")  
print(cart.total_cost()) # Should return 0.0
```

Output:

```
1.5  
2.5  
2.0  
2.0  
0
```

Justification:

The `ShoppingCart` class manages a collection of items and their prices. The `add_item` method adds an item to the cart, updating the price if the item already exists. The `remove_item` method removes an item from the cart, returning a message if the item is not found. The `total_cost` method calculates the total cost of all items in the cart by summing their prices. The test cases demonstrate adding items, calculating total cost, and removing items while validating the expected outcomes.

Task 5: Date Format Conversion

Scenario

You are creating a utility function to convert date formats for reports.

Requirements

- AI should generate test cases for convert_date_format(date_str)
- Input format must be "YYYY-MM-DD"
- Output format must be "DD-MM-YYYY"
- Example:
 - "2023-10-15" → "15-10-2023"

Expected Output

- Date conversion function implemented in Python
- Correct format conversion for all valid inputs
- All AI-generated test cases pass successfully

Prompt:

#Generate the code for function to convert date formats for reports. Input format must be "YYYY-MM-DD" and Output format must be "DD-MM-YYYY".

Code:

```
def convert_date_format(date_str: str) -> str:
    try:
        year, month, day = date_str.split('-')
        return f"{day}-{month}-{year}"
    except ValueError:
        return "Invalid date format"
# Test cases
print(convert_date_format("2024-06-15")) # Should return "15-06-2024"
print(convert_date_format("2024/06/15")) # Should return "Invalid date format"
print(convert_date_format("15-06-2024")) # Should return "Invalid date"
print(convert_date_format("2024-13-01")) # Should return "01-13-2024" (Note: This does not validate the date itself)
print(convert_date_format("2024-00-01")) # Should return "01-00-2024" (Note: This does not validate the date itself)
print(convert_date_format("2024-06-31")) # Should return "31-06-2024" (Note: This does not validate the date itself)
```

Output:

```
15-06-2024
Invalid date format
2024-06-15
01-13-2024
01-00-2024
31-06-2024
```

Justification:

The function `convert_date_format` takes a date string as input and attempts to convert it from the format "YYYY-MM-DD" to "DD-MM-YYYY". It first splits the input string into its components (year, month, and day) using the `split` method. Then, it rearranges these components into the desired output format using an f-string. If the input date string does not match the expected format, a `ValueError` will be raised, which is caught in the except block, returning an error message indicating the correct format to use. The test cases demonstrate both valid and invalid inputs to ensure the function behaves as expected.