ASSIGNMENT- 8.3

Name: M.Nishitha

HT. No: 2303A51540

Batch: 19

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

CODE:

```python
Users > nishithamarepally > nlp > email.py > validate_email
3    def validate_email(email):
5        regex = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
6
7        # If the email matches the regex pattern, it's valid
8        if re.match(regex, email):
9            return True
10       else:
11           return False
12
13
14   # User input
15   user_input = input("Enter an email address to validate: ")
16
17   # Output
18   print(validate_email(user_input))
19
```

OUTPUT:

```
  IndentationError: expected an indented block
● nishithamarepally@nishithas-MacBook-Air ~ % /usr/bin/python3 /Users/nishithamarepally/nlp/email.py
  Enter an email address to validate: nishithamarepally19@gmail.com
  True
○ nishithamarepally@nishithas-MacBook-Air ~ % ▋
```

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

CODE:

```
39
40   #generate automated grading system for an online examination platform.
41   # AI should generate test cases for assign_grade(score) where:
42   # 90-100 → A
43   # 80-89 → B
44   # 70-79 → C
45   # 60-69 → D
46   # Below 60 → F
47   # Include boundary values (60, 70, 80, 90)
48   # Include invalid inputs such as -5, 105, "eighty"
49   # Implement the function using a test-driven approach
50   def assign_grade(score):
51       if isinstance(score, str) or score < 0 or score > 100:
52           return "Invalid input"
53       if score >= 90:
54           return 'A'
55       elif score >= 80:
56           return 'B'
57       elif score >= 70:
58           return 'C'
59       elif score >= 60:
60           return 'D'
61       else:
62           return 'F'
63   # Test cases
64   test_scores = [95, 85, 75, 65, 55, 60, 70, 80, 90, -5, 105, "eighty"]
65   for score in test_scores:
66       print(f"{score}: {assign_grade(score)}")
67   #Expected Output: Correct grade for valid scores, "Invalid input" for invalid ones.
68
```

OUTPUT:

```
95: A
85: B
75: C
65: D
55: F
60: D
70: C
80: B
90: A
-5: Invalid input
105: Invalid input
eighty: Invalid input
```

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

CODE:

```python
#Generate a text-processing utility to analyze sentences.
# 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

import re
def is_sentence_palindrome(sentence):
    # Remove non-alphanumeric characters and convert to lowercase
    cleaned = re.sub(r'[^A-Za-z0-9]', '', sentence).lower()
    return cleaned == cleaned[::-1]
# Test cases
test_sentences = [
    "A man a plan a canal Panama",  # Palindrome
    "Hello World",                  # Not a palindrome
    "Was it a car or a cat I saw?", # Palindrome
    "No 'x' in Nixon",              # Palindrome
    "This is not a palindrome",     # Not a palindrome
    "Madam In Eden, I'm Adam",      # Palindrome
    "12321",                        # Palindrome
    "12345"                         # Not a palindrome
]
for sentence in test_sentences:
    print(f"'{sentence}': {is_sentence_palindrome(sentence)}")
#Expected Output: True for palindromic sentences, False for non-palindromic ones.
```

OUTPUT:

```
'A man a plan a canal Panama': True
'Hello World': False
'Was it a car or a cat I saw?': True
'No 'x' in Nixon': True
'This is not a palindrome': False
'Madam In Eden, I'm Adam': True
'12321': True
'12345': False
```

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

CODE:

```python
#Gnerate a basic shopping cart module for an e-commerce application.
# 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

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())
# Test cases
cart = ShoppingCart()
cart.add_item("Book", 12.99)
cart.add_item("Pen", 1.50)
print(f"Total cost after adding items: {cart.total_cost()}")  # Expected: 14.49
cart.remove_item("Pen")
print(f"Total cost after removing Pen: {cart.total_cost()}")  # Expected: 12.99
cart.remove_item("Notebook")  # Removing non-existing item
print(f"Total cost after trying to remove non-existing item: {cart.total_cost()}")  # Expected: 12.99
cart.remove_item("Book")
print(f"Total cost after removing Book: {cart.total_cost()}")  # Expected: 0.0
```

OUTPUT:

```
Total cost after adding items: 14.49
Total cost after removing Pen: 12.99
Total cost after trying to remove non-existing item: 12.99
Total cost after removing Book: 0
```

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

CODE:

```python
#Generate utility function to convert date formats for reports.
# 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"
from datetime import datetime

def convert_date_format(date_str):
    try:
        date_obj = datetime.strptime(date_str, "%Y-%m-%d")
        return date_obj.strftime("%d-%m-%Y")

    except ValueError:
        return "Invalid date format"

# Test cases
test_dates = [
    "2023-10-15",   # Valid date
    "1990-01-01",   # Valid date
    "2023/10/15",   # Invalid format
    "15-10-2023",   # Invalid format
    "2023-13-01",   # Invalid month
    "2023-00-10",   # Invalid month
    "2023-10-32",   # Invalid day
    "2023-10-00"    # Invalid day
]
for date in test_dates:
    print(f"{date}: {convert_date_format(date)}")
#Expected Output: Correctly converted date for valid inputs, "Invalid date format" for invalid ones.
```

OUTPUT:

```
2023-10-15: 15-10-2023
1990-01-01: 01-01-1990
2023/10/15: Invalid date format
15-10-2023: Invalid date format
2023-13-01: Invalid date format
2023-00-10: Invalid date format
2023-10-32: Invalid date format
2023-10-00: Invalid date format
```