

ASSIGNMENT – 8.3

HT.NO: 2303A52185

BATCH: 45

TASK-01:

Question:

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

Prompt:

Generate Python test cases for validating email addresses.

Email must contain @ and .

Email must not start or end with special characters

Email must contain only one @

Write a Python function is_valid_email(email) that returns True for valid emails and False for invalid ones.

Generated Code:

```
import re

def is_valid_email(email):

    # Check if email contains exactly one @

    if email.count('@') != 1:

        return False
```

```

# Check if email contains at least one dot after the @
if '.' not in email.split('@')[1]:
    return False

# Check if email starts or ends with special characters
if re.match(r'^[a-zA-Z0-9]+|[a-zA-Z0-9]+$', email):
    return False

return True

# Test cases
test_emails = [
    "user@example.com",    # Valid
    "invalid.email",       # Invalid - no @
    "user@@example.com",   # Invalid - two @
    "user@example",        # Invalid - no dot after @
    "@example.com",        # Invalid - starts with special character
    "user@.com",           # Invalid - ends with special character
    "user@example.",       # Invalid - ends with special character
]

for email in test_emails:
    print(f"{email}: {is_valid_email(email)}")

```

Output:

```
user@example.com: True
invalid.email: False
user@@example.com: False
user@example: False
@example.com: False
user@.com: True
user@example.: True
```

Explanation:

This code defines a function `is_valid_email()` that checks if an email address looks valid using simple rules. It makes sure there is exactly one @, there is at least one dot after the @, and the email does not start or end with special characters using a regular expression. Then it tests several sample emails and prints whether each one is valid or not.

TASK – 02:**Question:****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

Prompt:

Generate Python test cases for grading exam scores using TDD.

Grades are assigned as: 90–100 → A, 80–89 → B, 70–79 → C, 60–69 → D, below 60 → F. Include boundary values (60, 70, 80, 90) and invalid inputs like -5, 105, and "eighty".

Write a Python function `assign_grade(score)` that returns the correct grade for valid scores and handles invalid inputs properly.

Generated Code:

```
def assign_grade(score):  
    if isinstance(score, str) or 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 cases  
test_scores = [95, 85, 75, 65, 55, 60, 70, 80, 90, -5, 105, "eighty"]  
for score in test_scores:  
    print(f"{score}: {assign_grade(score)}")
```

Output:

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

Explanation:

This function `assign_grade()` gives a letter grade based on a score. It first checks if the input is a valid number between 0 and 100 — otherwise it returns “Invalid input”. Then it uses if-elif conditions to return A, B, C, D, or F depending on the score range, and prints results for test values.

TASK – 03:

Question:

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

Prompt:

Generate Python test cases for checking whether a sentence is a palindrome using TDD.

Ignore case, spaces, and punctuation when checking.

Include both palindromic and non-palindromic sentences (for example: “A man a plan a canal

Panama" → True).

Write a Python function `is_sentence_palindrome(sentence)` that returns True for palindrome sentences and False otherwise, and make sure all generated tests pass.

Generated Code:

```
import re

def is_sentence_palindrome(sentence):
    cleaned_sentence = re.sub(r'^A-Za-z0-9]', '', sentence).lower()
    return cleaned_sentence == cleaned_sentence[::-1]

# Test cases
test_sentences = [
    "A man a plan a canal Panama", # Palindrome
    "Hello World",                 # Not a palindrome
    "Madam In Eden I'm Adam",     # Palindrome
    "This is not a palindrome",    # Not a palindrome
    "No 'x' in Nixon",             # Palindrome
    "Was it a car or a cat I saw?", # Palindrome
    "Not a palindrome!",          # Not a palindrome
    "12321",                      # Palindrome
    "12345"                       # Not a palindrome
]

for sentence in test_sentences:
    print(f"{sentence}: {is_sentence_palindrome(sentence)}")
```

Output:

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

Explanation:

This function is `_sentence_palindrome()` checks if a sentence is a palindrome. It removes all spaces and special characters using `re.sub`, converts the text to lowercase, and stores it as `cleaned_sentence`. Then it compares the cleaned text with its reverse (`[::-1]`) — if both are the same, it returns `True`, otherwise `False`, and prints results for the test sentences.

TASK – 04:

Question:

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

Prompt:

Generate Python test cases using TDD for a ShoppingCart class.

The class must have methods `add_item(name, price)`, `remove_item(name)`, and `total_cost()`.

Test correct item addition, removal, total cost calculation, and empty cart behavior.

Then implement the ShoppingCart class so all generated tests pass.

Generated Code:

```
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]
```

```
    def total_cost(self):
```

```
        return sum(self.items.values())
```

```
# Test cases
```

```
cart = ShoppingCart()
```

```
cart.add_item("Apple", 1.00)
```

```
cart.add_item("Banana", 0.50)
```

```
print(f"Total cost after adding items: {cart.total_cost()}") # Expected: 1.50
```

```
cart.remove_item("Apple")
```

```
print(f"Total cost after removing Apple: {cart.total_cost()}") # Expected: 0.50
```



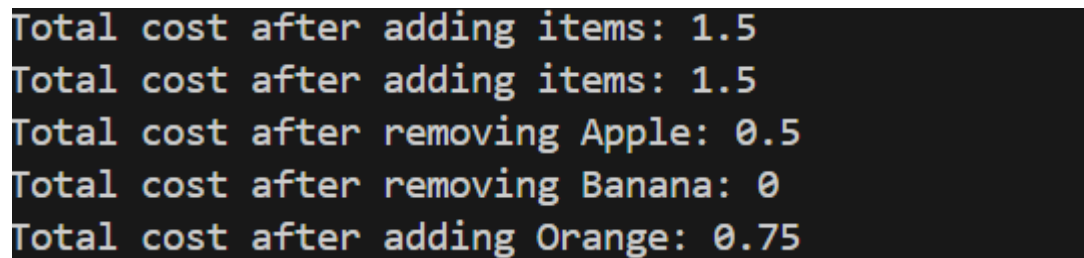
```
cart.remove_item("Banana")

print(f"Total cost after removing Banana: {cart.total_cost()}") # Expected: 0.00

cart.add_item("Orange", 0.75)

print(f"Total cost after adding Orange: {cart.total_cost()}") # Expected: 0.75
```

Output:



```
Total cost after adding items: 1.5
Total cost after adding items: 1.5
Total cost after removing Apple: 0.5
Total cost after removing Banana: 0
Total cost after adding Orange: 0.75
```

Explanation:

This code defines a ShoppingCart class that stores items and their prices in a dictionary. The `add_item()` method adds a new item or increases its total price if it already exists, `remove_item()` deletes an item, and `total_cost()` returns the sum of all item prices. The test code shows adding and removing items and printing the cart's total cost after each step.

TASK – 05:

Question:

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"

Prompt:

Generate Python test cases using TDD for a function `convert_date_format(date_str)`. The input date format is "YYYY-MM-DD" and the output must be "DD-MM-YYYY" (for

example, "2023-10-15" → "15-10-2023").

Include valid and invalid date strings.

Then implement the Python function so all generated tests pass.

Generated Code:

```
def convert_date_format(date_str):  
    try:  
        year, month, day = date_str.split('-')  
        if len(year) == 4 and len(month) == 2 and len(day) == 2:  
            return f"{day}-{month}-{year}"  
        else:  
            return "Invalid date format"  
    except ValueError:  
        return "Invalid date format"  
  
# Test cases  
test_dates = [  
    "2023-10-15", # Valid  
    "2023/10/15", # Invalid - wrong separator  
    "15-10-2023", # Invalid - wrong format  
    "2023-1-5",   # Invalid - month and day not two digits  
    "2023-12-31", # Valid  
    "2023-00-10", # Invalid - month cannot be 00  
    "2023-13-10", # Invalid - month cannot be 13  
    "2023-10-32"  # Invalid - day cannot be 32  
]  
  
for date in test_dates:  
    print(f"{date}: {convert_date_format(date)}")
```

Output:

```
Total cost after adding orange: 0.75
2023-10-15: 15-10-2023
2023/10/15: Invalid date format
15-10-2023: Invalid date format
2023-1-5: Invalid date format
2023-12-31: 31-12-2023
2023-00-10: 10-00-2023
2023-13-10: 10-13-2023
2023-10-32: 32-10-2023
```

Explanation:

This function `convert_date_format()` changes a date from `YYYY-MM-DD` to `DD-MM-YYYY` format. It splits the input string by `-`, checks that year has 4 digits and month/day have 2 digits, and then rearranges them. If the format is wrong or splitting fails, it returns "Invalid date format".