

# ASSIGNMENT-13.4

**SHIVANI PAMBIDI**

**2303A52172**

**BATCH-41**

## QUESTION 1

### TASK 1: Refactoring Data Transformation

**PROMPT:** Refactor the code using list comprehensions or helper functions while preserving the output.

**Legacy Code:**

```
Lab 13.4.py > ...
1  values = [2, 4, 6, 8, 10]
2  doubled = []
3  for v in values:
4      doubled.append(v * 2)
5  print(doubled)
```

**Refactored Code:**

```
6  """
7  """
8  This script doubles each number in a list using a helper function.
9  The helper function is applied to every element using list comprehension.
10 """
11 def double(x):
12     """
13         This function takes a number as input
14         and returns the number multiplied by 2.
15     """
16     return x * 2
17 values = [2, 4, 6, 8, 10]
18 # Apply the double() function to each value in the list
19 doubled = [double(v) for v in values]
20
21 # Print the result
22 print(doubled)
```

## OUTPUT:

```
● PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assisted-Coding/Lab 13.4.py"
[4, 8, 12, 16, 20]
○ PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding>
```

## JUSTIFICATION:

The new code is shorter and cleaner. It is easier to read and understand. Python supports list comprehension, so it is a better way to write this type of code. The output is the same as the old code. It reduces extra lines and makes the program simple. It avoids creating an empty list and manually adding elements one by one. It makes the logic clear in a single line (multiply each value by 2). It follows modern Python coding style, which is preferred in real projects.

## QUESTION 2

### TASK 2: Improving Text Processing Code Readability

#### PROMPT:

Refactor the following Python code to improve efficiency and readability. Replace repeated string concatenation with a better Python method while keeping the same output.

#### Legacy CODE:

```
22     print(message)
23
24     words = ["Refactoring", "with", "AI", "improves",
25             "quality"]
26     message = ""
27     for w in words:
28         message += w + " "
29     print(message.strip())
```

#### Refactored Code:

```
30 """
31 """
32 This script creates a sentence from a list of words.
33
34 Instead of adding words one by one using string concatenation (+),
35 it uses the join() method which is more efficient and readable.
36
37 The final output remains the same as the legacy code.
38 """
39
40 # List of words to combine
41 words = ["Refactoring", "with", "AI", "improves", "quality"]
42
43 # Use join() to combine all words into one string with spaces
44 message = " ".join(words)
45
46 # Print the final sentence
47 print(message)
```

## OUTPUT:

```
● PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> & C:/Users/pambi/AppData/thon313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assisted-Coding/Lab 13.4 Refactoring with AI improves quality"
○ PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> []
Ln 47 Col 15  Spaces: 4  LITE 8  CRLE { 1 Path}
```

## JUSTIFICATION:

The refactored code uses the join() method instead of adding strings one by one using +=. The join() method is faster and more efficient because it avoids creating many temporary strings. It makes the code shorter and easier to read. The logic is clear since all words are combined in a single line. It follows good Python coding practices. The final output remains the same as the original code.

## QUESTION 3

### TASK 3: Safer Access to Configuration Data

#### PROMPT:

#### Legacy CODE:

```
48
49     config = {"host": "localhost", "port": 8080}
50     if "timeout" in config:
51         print(config["timeout"])
52     else:
53         print("Default timeout used")
```

#### Refactored Code:

```
"""
This script safely accesses a dictionary value using the get() method.
If the key does not exist, it prints a default message.
"""

config = {"host": "localhost", "port": 8080}

# Safely get the value of "timeout"
timeout = config.get("timeout")

if timeout is not None:
    print(timeout)
else:
    print("Default timeout used")
```

## OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> & C:/Users/pambi/AppData/Local/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assisted-Coding/Lab 13.4.py"
Default timeout used
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> []
```

## JUSTIFICATION:

The refactored code uses the get() method to safely access values from the dictionary. It removes the need to manually check whether the key exists using if "key" in dictionary. This makes the code shorter and cleaner. The get() method prevents a KeyError if the key is missing. It improves code readability because the intention is clear. It reduces the number of lines compared to the manual checking approach. It is a safer and more professional way to handle configuration data. It works well in real-world applications where some settings may not always be present. The overall behavior of the program remains the same. If the key is missing, the default message is still printed correctly.

## QUESTION 4

### TASK 4: Refactoring Conditional Logic for Scalability

#### PROMPT:

Refactor the following Python code by replacing multiple if–elif conditions with a cleaner and more scalable approach using mapping techniques. Make sure the functionality and output remain the same.

#### Legacy CODE:

```
59     action = "divide"
60     x, y = 10, 2
61
62     if action == "add":
63         result = x + y
64     elif action == "subtract":
65         result = x - y
66     elif action == "multiply":
67         result = x * y
68     elif action == "divide":
69         result = x / y
70     else:
71         result = None
72
73     print(result)
```

## Refactored Code:

```
76 """
77     This script performs different mathematical operations
78     based on user input using dictionary mapping instead of
79     multiple if-elif conditions.
80 """
81
82
83     action = "divide"
84     x, y = 10, 2
85
86     # Mapping actions to corresponding operations
87     operations = {
88         "add": lambda a, b: a + b,
89         "subtract": lambda a, b: a - b,
90         "multiply": lambda a, b: a * b,
91         "divide": lambda a, b: a / b
92     }
93
94     # Get the correct function from dictionary and execute it
95     result = operations.get(action, lambda a, b: None)(x, y)
96
97     print(result)
```

## OUTPUT:

```
● PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> & C:/Users/pambi/AppData/Local/thon313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assisted-Coding/Lab 13.4.py"
5.0
○ PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> □
```

## JUSTIFICATION:

The refactored code replaces multiple if–elif statements with a dictionary mapping. This makes the code shorter and more organized. Adding a new operation is easier because we just add one new entry in the dictionary. It improves scalability since the logic does not become messy with many conditions. The code becomes more readable and structured. It follows a cleaner and more modern Python approach. The functionality and output remain exactly the same as the original code.

## QUESTION 5

### TASK 5: Simplifying Search Logic in Collection

#### PROMPT:

Refactor the following Python code to simplify the search logic. Replace the manual loop with a more concise and readable method while keeping the same output behavior.

### Legacy Code:

```
8
9     inventory = ["pen", "notebook", "eraser", "marker"]
10    found = False
11    for item in inventory:
12        if item == "eraser":
13            found = True
14            break
15    print("Item Available" if found else "Item Not Available")
```

### Refactored Code:

```
"""
This script checks whether an item exists in the inventory list.
Instead of using a manual loop, it uses the 'in' operator
to make the code shorter and more readable.
"""

inventory = ["pen", "notebook", "eraser", "marker"]

# Check directly if "eraser" is in the list
found = "eraser" in inventory

print("Item Available" if found else "Item Not Available")
```

### OUTPUT:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding> & C:/Users/pambi/AppData/Local/Programs/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assisted-Coding/Lab 13.4.py"
Item Available
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assisted-Coding>
```

### JUSTIFICATION:

The refactored code uses the `in` operator instead of a manual loop. It removes the need for a flag variable and `break` statement. The code becomes shorter and easier to understand. The intention of checking an item is clearly visible in one line. It follows better Python coding practice. The output remains exactly the same as the original code. It improves readability and makes maintenance easier in the future. It reduces the chances of logical errors in loop handling. It makes the program more efficient and clean for larger collections.