T.Akhil
2303A53024

AssignmentNumber:13.4 (Present assignment number)/**24**(Total number of assignments)

Q.No.	Question	*ExpectedTime to complete*
1	**Lab 13: Code Refactoring – Improving Legacy Code with AI Suggestions** **Lab Objectives** • Identify code smells and inefficiencies in legacy Python scripts. • Use AI-assisted coding tools to **refactor** for readability, maintainability, and performance. • Apply **modern Python best practices** while ensuring output correctness. **Learning Outcome** After completing this assignment, students will be able to: • Identify refactoring opportunities in legacy Python code • Apply AI-assisted suggestions to refactor code without changing behavior • Refactor Python code using Pythonic and scalable constructs • Critically evaluate AI-generated refactoring recommendations	Week 7
	# Task 1: Refactoring Data Transformation Logic ## Scenario You are maintaining a **data preprocessing script** where numerical transformations are written using verbose loops. ## Task Description • Review the legacy code that computes transformed values using an explicit loop. • Use an AI tool to suggest a more Pythonic refactoring approach.	

- Refactor the code using list comprehensions or helper functions while preserving the output.

Legacy Code

```
values = [2, 4, 6, 8, 10]
doubled = []
for v in values:
    doubled.append(v * 2)
print(doubled)
```

Expected Output

```
[4, 8, 12, 16, 20]
```

Prompt:

```
I have legacy Python code that uses a loop to transform
data.

Code:
values = [2, 4, 6, 8, 10]
doubled = []
for v in values:
    doubled.append(v * 2)

Refactor this code using a more Pythonic approach like
list comprehension or helper functions.
Keep the output exactly the same.
Explain why the refactored version is better in terms of
readability and performance.
```

Task 2: Improving Text Processing Code Readability

Scenario

You are working on a **log message generator** that builds messages word by word.

Task Description

- Analyze the legacy code that constructs a sentence using repeated string concatenation.
- Ask AI to suggest a more efficient and readable approach.

- Refactor the code accordingly while keeping the final output unchanged.

Legacy Code

```
words = ["Refactoring", "with", "AI", "improves",
"quality"]
message = ""
for w in words:
    message += w + " "
print(message.strip())
```

Expected Output

```
Refactoring with AI improves quality
```

Prompt:

```
I have legacy Python code that builds a sentence using
string concatenation inside a loop.

Code:
words = ["Refactoring", "with", "AI", "improves",
"quality"]
message = ""
for w in words:
    message += w + " "

Refactor this code into a cleaner and more efficient
approach.
Do not change the final output.
Explain why your method is more efficient and readable.
```

Task 3: Safer Access to Configuration Data

Scenario

You are maintaining a **configuration loader** where dictionary keys may or may not exist depending on deployment settings.

Task Description

- Review the legacy code that manually checks for dictionary keys.
- Use AI suggestions to refactor the code using safer dictionary access methods.
- Ensure the behavior remains the same for missing keys.

Legacy Code

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

if "timeout" in config:
    print(config["timeout"])
else:
    print("Default timeout used")
```

Expected Output

```
Default timeout used
```

Prompt:

```
I have Python code that manually checks if a dictionary
key exists.

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

if "timeout" in config:
    print(config["timeout"])
else:
    print("Default timeout used")

Refactor this code using safer dictionary access methods
(like get()).
Ensure the behavior remains the same when keys are
missing.
Explain why the new approach is safer.
```

Task 4: Refactoring Conditional Logic for Scalability

Scenario

You are enhancing a **utility module** that performs different operations based on user input.

Task Description

- Examine the multiple if–elif conditions used to determine operations.
- Ask AI to suggest a cleaner, scalable alternative.

- Refactor the logic using mapping techniques while preserving functionality.

Legacy Code

```
action = "divide"
x, y = 10, 2

if action == "add":
    result = x + y
elif action == "subtract":
    result = x - y
elif action == "multiply":
    result = x * y
elif action == "divide":
    result = x / y
else:
    result = None

print(result)
```

Expected Output

```
5.0
```

Prompt:

```
I have Python code using multiple if-elif conditions
to perform operations.

Code:
action = "divide"
x, y = 10, 2

if action == "add":
    result = x + y
elif action == "subtract":
    result = x - y
elif action == "multiply":
    result = x * y
elif action == "divide":
    result = x / y
else:
    result = None

Refactor this using a scalable and cleaner approach
such as mapping or dictionaries.
Keep functionality unchanged.
Explain why the new design is better for scalability.
```

Task 5: Simplifying Search Logic in Collections

Scenario

You are reviewing **legacy inventory-check code** that uses manual loops to find elements.

Task Description

- Identify the explicit loop used for searching an item.
- Use AI assistance to refactor the logic into a more concise and readable form.
- Maintain the same output behavior.

Legacy Code

```
inventory = ["pen", "notebook", "eraser", "marker"]
found = False

for item in inventory:
    if item == "eraser":
        found = True
        break

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

Expected Output

```
Item Available
```

Prompt:

```
I have legacy Python code that searches for an item
using a manual loop.

Code:
inventory = ["pen", "notebook", "eraser", "marker"]
found = False

for item in inventory:
    if item == "eraser":
        found = True
        break
```

	Refactor this code into a shorter and more Pythonic solution. Do not change the output behavior. Explain why your solution improves readability.	