SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name	Venkataramana Veeramsetty		
Instructor(s) Name	Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)		
Course Code	24CS002PC215	**Course Title**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
Date and Day of Assignment	Week7 - Thursday	**Time(s)**	
Duration	2 Hours	**Applicable to Batches**	
AssignmentNumber:13.1(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*Expected Time 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,	Week7 - Thursday

maintainability, and performance.

- Apply **modern Python best practices** while ensuring output correctness.

Task 1

- **Task:** Refactor repeated loops into a cleaner, more Pythonic approach.

Instructions:

- Analyze the legacy code.
- Identify the part that uses loops to compute values.
- Refactor using **list comprehensions** or helper functions while keeping the output the same.

Legacy Code:

```
numbers = [1, 2, 3, 4, 5]
squares = []
for n in numbers:
    squares.append(n ** 2)
print(squares)
```

Expected Output:

```
[1, 4, 9, 16, 25]
```

CODE:

```
<-1.py > ...
"""Compute squares of a list using a Pythonic list comprehension.

This module defines a helper function `square_all` and demonstrates refactoring
an explicit append loop into a list comprehension while preserving output.

Expected output:
    [1, 4, 9, 16, 25]
"""
def square_all(values):
    return [v ** 2 for v in values]
numbers = [1, 2, 3, 4, 5]
squares = square_all(numbers)
print(squares)
```

OUTPUT:

```
PS C:\Users\DEEKSHA\OneDrive\Desktop\AIAC\Lab-13.1
[1, 4, 9, 16, 25]
```

Task 2

Task: Simplify string concatenation.

Instructions:

- Review the loop that builds a sentence using +=.
- Refactor using " ".join() to improve efficiency and readability.

Legacy Code:

words = ["AI", "helps", "in", "refactoring", "code"]
sentence = ""
for word in words:
    sentence += word + " "
print(sentence.strip())

Expected Output:

AI helps in refactoring code

CODE:

```python
"""Build a sentence from words using a Pythonic join operation.

This module demonstrates refactoring a string-building loop with += into a
single " ".join(...) call for clarity and efficiency.

Expected output:
    AI helps in refactoring code
"""

def build_sentence(words):
    return " ".join(words)

words = ["AI", "helps", "in", "refactoring", "code"]
sentence = build_sentence(words)
print(sentence)
```

OUTPUT:

```
AI helps in refactoring code
```

-

Task 3

Task: Replace manual dictionary lookup with a safer method.

Instructions:

- Check how the code accesses dictionary keys.
- Use .get() or another Pythonic approach to handle missing keys gracefully.

Legacy Code:

student_scores = {"Alice": 85, "Bob": 90}
if "Charlie" in student_scores:
    print(student_scores["Charlie"])
else:
    print("Not Found")

Expected Output:

Not Found

CODE:

```
"""Safely access dictionary keys using .get() with a default value.

This script shows a Pythonic alternative to explicit membership checks by
using dict.get(key, default) to handle missing keys gracefully.

Expected output when 'Charlie' is missing:
    Not Found
"""

student_scores = {"Alice": 85, "Bob": 90}
print(student_scores.get("Charlie", "Not Found"))
```

OUTPUT:

```
PS C:\Users\DEEKSHA\OneDrive\Desktop\AIAC\Lab-13.1
Not Found
```

Task 4

Task: Refactor repetitive if-else blocks.

Instructions:

- Examine multiple if-elif statements for operations.
- Refactor using **dictionary mapping** to make the code scalable and clean.

Legacy Code:

```
operation = "multiply"
a, b = 5, 3

if operation == "add":
    result = a + b
elif operation == "subtract":
    result = a - b
elif operation == "multiply":
    result = a * b
else:
    result = None

print(result)
```

Expected Output:

15

CODE:

```
"""Select arithmetic operations via a dictionary-based dispatch.

This script replaces an if-elif chain with a mapping from operation names to
callables, making it easy to add new operations while preserving behavior.

Expected output for operation "multiply" and inputs 5, 3:
    15
"""

operation = "multiply"
a, b = 5, 3

operations = {
    "add": lambda x, y: x + y,
    "subtract": lambda x, y: x - y,
    "multiply": lambda x, y: x * y,
}

func = operations.get(operation)
result = func(a, b) if func else None

print(result)
```

OUTPUT:

```
15
PS C:\Users\DEEKSHA\OneDrive\Desktop\AIAC\L
```

Task 5

Task: Optimize nested loops for searching.

Instructions:

- Identify the nested loop used to find an element.
- Refactor using Python's in keyword or other efficient search techniques.

Legacy Code:

```
items = [10, 20, 30, 40, 50]
found = False
for i in items:
    if i == 30:
        found = True
        break
print("Found" if found else "Not Found")
```

Expected Output:

Found

CODE:

```
"""Search for an element using Pythonic containment instead of a manual loop.

This script replaces a nested/explicit search loop with the `in` keyword,
which is more readable and efficient for membership checks.

Expected output when searching for 30 in the list:
    Found
"""

items = [10, 20, 30, 40, 50]
found = 30 in items
print("Found" if found else "Not Found")
```

OUTPUT:

```
PS C:\Users\DEEKSHA\OneDrive\Desktop\AIAC\Lab-13.1>
Found
```