**ASSIGNMENT-13**

**H NO: 2403A52139**

* **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:

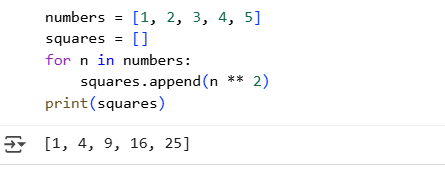
squares.append(n \*\* 2)

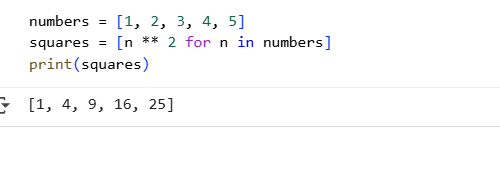
print(squares)

**Expected Output:**

[1, 4, 9, 16, 25]

* **CODES AND OUTPUTS**

****

****

**EXPLANATION:**

* For Loop: Explicitly initializes an empty list and uses append() inside a loop to add elements one by one. It's more verbose but can handle complex logic within the loop.
* List Comprehension: Creates a new list in a single line using a concise syntax [expression for item in iterable]. It's generally more readable and faster for simple list transformations.

Both methods achieve the same result for this task, but list comprehensions are often preferred in Python for their brevity and performance when applicable.

* **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:

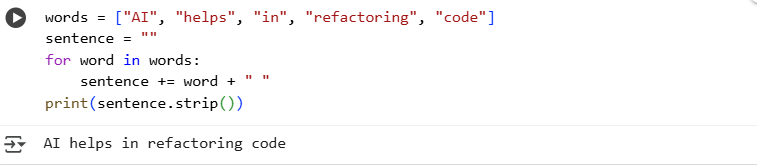
sentence += word + " "

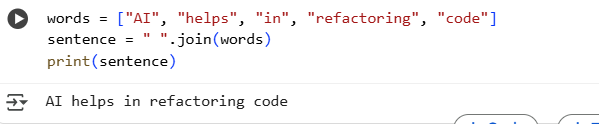
print(sentence.strip())

**Expected Output:**

AI helps in refactoring code

**CODES AND OUTPUTS**





**EXPLANATION**

* **+= in loop:** More explicit step-by-step concatenation, can be less efficient for many concatenations. Requires strip() to clean up the end.
* **join():** More concise, efficient, and Pythonic for concatenating strings from an iterable with a specified separator. No need for strip() in this case as the separator is only placed *between* elements.

Both produce the same output, but join() is the preferred method in Python for this type of task due to its performance and readability benefits.

* **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"])

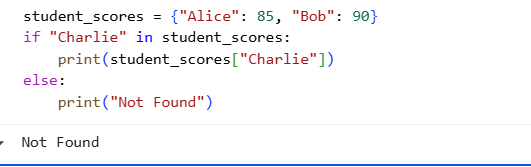
else:

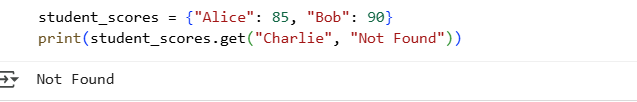
print("Not Found")

**Expected Output:**

Not Found

* **CODES AND OUTPUTS**

****



**EXPLANATION**

* Use the if 'key' in dictionary: and dictionary['key'] approach when you need to perform different actions based on whether a key exists or when you want to explicitly handle the KeyError yourself.
* Use the .get() method when you simply want to retrieve a value if a key exists, or get a default value if it doesn't, without the risk of a KeyError.

For simply retrieving a value with a fallback, the .get() method is generally considered more "Pythonic" due to its conciseness and error prevention.

* **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

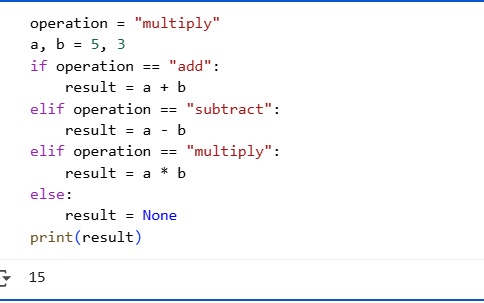
else:

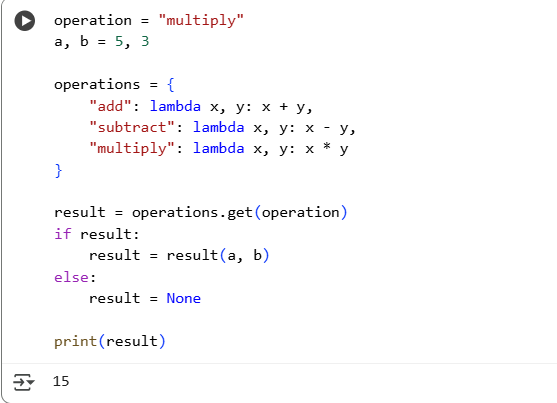
result = None

print(result)

**Expected Output:**15

**CODES AND OUTPUT**





**EXPLANATION**

**Key Differences**

* Structure: The first uses a linear series of checks (if/elif), while the second uses a dictionary for a direct mapping.
* Scalability: Adding a new operation in the original code requires adding another elif block. In the refactored code, you just add a new entry to the operations dictionary. This is much cleaner for many operations.
* Readability: For a small number of operations, the if-elif-else might be clear. However, for a larger number, the dictionary mapping is much more organized and easier to read.
* Maintainability: Changes or additions to operations are localized within the operations dictionary in the refactored code, making it easier to maintain.

Both achieve the same outcome for this specific example, but the dictionary mapping approach is generally preferred for handling multiple distinct actions based on a key or command due to its scalability and cleaner structure.

* **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

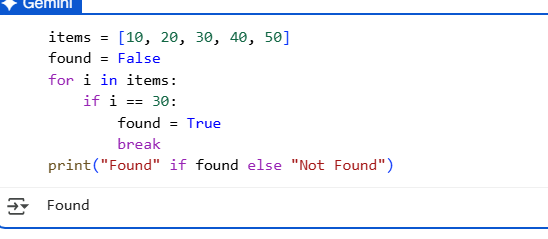
break

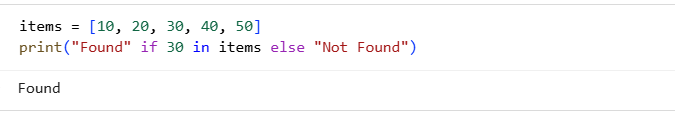
print("Found" if found else "Not Found")

**Expected Output:**

Found

**CODES AND OUTPUT**

****

****

**EXPLANATION**

* The loop-based approach is more explicit and gives you fine-grained control over the iteration process, allowing for more complex actions within the loop.
* The in operator is the Pythonic way for simple membership checking. It's more concise, readable, and efficient for this specific task.

For simply determining if an item exists in a list, the in operator is almost always the preferred method in Python**.**