**AI ASSISSTED CODING-13:**

**TASK-1:**

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)

A screenshot of a math problem

AI-generated content may be incorrect.

A white background with numbers and letters

AI-generated content may be incorrect.

**EXPLANATION:**

The first code cell with ID MOGceSCKcvlR uses a **list comprehension** to create a new list called squares.

* numbers = [1, 2, 3, 4, 5]: This line creates a list named numbers containing integers from 1 to 5.
* squares = [n \*\* 2 for n in numbers]: This is the list comprehension. It iterates through each element (n) in the numbers list and creates a new list where each element is the square of n (n \*\* 2).
* print(squares): This line prints the resulting squares list.

The second code cell with ID 4MYvRCKvddix achieves the same result using a traditional **for loop**.

* numbers = [1, 2, 3, 4, 5]: Similar to the first cell, this creates the numbers list.
* squares = []: An empty list called squares is initialized.
* for n in numbers:: This starts a for loop that iterates through each element (n) in the numbers list.
* squares.append(n \*\* 2): Inside the loop, the square of the current element (n \*\* 2) is calculated and appended to the squares list.
* print(squares): Finally, this line prints the squares list.

Both cells produce the same output: [1, 4, 9, 16, 25], but the first cell demonstrates a more concise way to achieve this using list comprehension, which is often considered more "pythonic".

**TASK-2:**

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())

A screenshot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**EXPLANATION:**

The code in cell wmXikd\_lfE5y constructs a sentence from a list of words using a for loop and string concatenation:

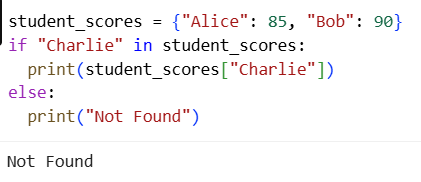
* words = ["AI", "helps", "in", "refactoring", "code"]: This line initializes a list named words containing several strings.
* sentence = "": An empty string variable sentence is created to store the resulting sentence.
* for word in words:: This loop iterates through each word in the words list.
* sentence += word + " ": Inside the loop, each word is appended to the sentence string, followed by a space. This builds the sentence word by word with spaces in between.
* print(sentence.strip()): Finally, print() displays the assembled sentence. The .strip() method is used to remove any leading or trailing whitespace from the sentence (in this case, it removes the extra space at the end of the sentence).

This code achieves the same result as the cell using "".join(), but the "".join() method is generally more efficient for concatenating many strings.

**TASK-3:**

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")

****

**A white background with text

AI-generated content may be incorrect.**

**EXPLANATION:**

Here's a breakdown:

* student\_scores = {"Alice": 85, "Bob": 90}: This line creates a dictionary named student\_scores. Dictionaries store data in key-value pairs. In this case, the keys are student names (strings) and the values are their scores (integers).
* print(student\_scores.get("Charlie", "Not Found")): This line uses the .get() method to retrieve a value from the student\_scores dictionary.
  + "Charlie" is the key we are trying to find in the dictionary.
  + "Not Found" is the default value. If the key "Charlie" is found in the dictionary, the method will return the corresponding value (which would be Charlie's score if he were in the dictionary). If "Charlie" is *not* found (as in this example), the method will return the default value "Not Found" instead of raising a KeyError.
  + print() then displays the value returned by the .get() method.

So, in this specific code, since "Charlie" is not a key in the student\_scores dictionary, the output will be "Not Found". Using .get() is generally preferred over directly accessing a key with square brackets (student\_scores["Charlie"]) when you're not sure if the key exists, as it prevents errors.

**TASK-4:**

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)

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**EXPLANATION:**

This code demonstrates a clean and scalable way to perform different operations based on a string input using dictionary mapping. Here's a breakdown:

1. **Function Definitions**:
   * def add(x, y): return x + y: Defines a function named add that takes two arguments (x and y) and returns their sum.
   * def subtract(x, y): return x - y: Defines a function named subtract that takes two arguments and returns their difference.
   * def multiply(x, y): return x \* y: Defines a function named multiply that takes two arguments and returns their product.
2. **Variables**:
   * operation = "multiply": A string variable operation is set to "multiply", indicating the desired operation.
   * a, b = 5, 3: Two integer variables a and b are assigned the values 5 and 3, respectively. These will be the operands for the operation.
3. **Dictionary Mapping**:
   * operations = { ... }: A dictionary named operations is created. This is the key part of the refactoring.
     + The keys of the dictionary are strings representing the names of the operations ("add", "subtract", "multiply").
     + The values of the dictionary are the actual function objects (add, subtract, multiply). This maps the string name to the function itself.
4. **Getting the Function**:
   * operation\_function = operations.get(operation): This line uses the .get() method of the operations dictionary to retrieve the function associated with the string stored in the operation variable ("multiply").
     + If the key ("multiply") is found in the dictionary, the corresponding function object (multiply) is assigned to operation\_function.
     + The .get() method is used with no second argument here, so if the key is not found, it will return None by default.
5. **Performing the Operation**:
   * if operation\_function:: This checks if operation\_function is not None (meaning the operation name was found in the dictionary).
     + result = operation\_function(a, b): If a function was found, it is called with a and b as arguments, and the returned value (the result of the operation) is stored in the result variable.
   * else:: If operation\_function is None (the operation name was not found).
     + result = None: The result is set to None.
     + print(f"Error: Operation '{operation}' not supported."): An informative error message is printed indicating that the requested operation is not supported.
6. **Printing the Result**:
   * print(result): Finally, the value stored in the result variable is printed. In this case, since operation is "multiply", the multiply function is called with 5 and 3, and the output will be 15.

This approach is much cleaner and more expandable than using a long chain of if/elif/else statements, especially when you have many possible operations. You can easily add new operations by defining a function and adding an entry to the operations dictionary

**TASK-5:**

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")

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

**EXPLANATION:**

Here's what this code does:

1. items = [10, 20, 30, 40, 50]: This line creates a list named items containing five integer values.
2. found = 30 in items: This is the core of the code. The in keyword in Python is used to check for membership in a sequence (like a list, tuple, string, or set). It checks if the value 30 exists within the items list. The result of this check is a boolean value (True if found, False if not found), which is then assigned to the variable found.
3. print("Found" if found else "Not Found"): This line uses a conditional expression (often called a ternary operator) to print a message based on the value of the found variable.
   * If found is True, it prints "Found".
   * If found is False, it prints "Not Found".

This code is a very Pythonic and efficient way to determine if a specific element exists within a list.