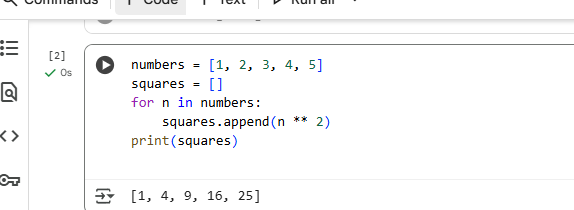
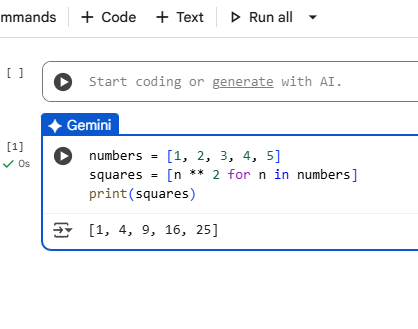
AI Assisted Coding

Lab Assingment-13

**Task 1**

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





Explaination:

* **Cell (Traditional for loop):** This uses a standard for loop to iterate through the numbers list. It initializes an empty list called squares and then appends the square of each number to this list one by one within the loop. This is a more explicit and step-by-step approach.
* **Cell  (List comprehension):** This uses a list comprehension, which is a more concise and Pythonic way to create lists. It combines the loop and the appending operation into a single line of code. It's often considered more readable and efficient for simple list creation tasks.

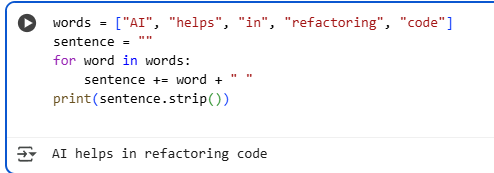
In essence, the list comprehension in cell f2914d1d is a more compact and often preferred way to write the same logic as the traditional for loop in cell Nf86nbxZBXiX when you're creating a new list based on an existing iterable.

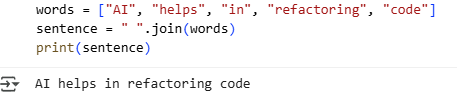
**Task 2**

**Task:** Simplify string concatenation.  
**Instructions:**

* Review the loop that builds a sentence using +=.

Refactor using " ".join() to improve efficiency and readability.





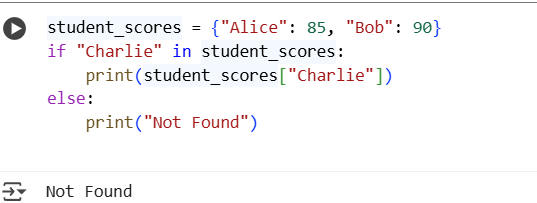
Explaination:

* **Cell  (Traditional for loop with +=):** This code uses a for loop to iterate through the words list. Inside the loop, it repeatedly uses the += operator to append each word and a space to the sentence string. While this works, it can be less efficient for a large number of words because creating new strings in each iteration can be computationally expensive.
* **Cell  ("".join() method):** This code uses the "".join() method. This method is generally more efficient for concatenating strings from a list or iterable. It takes an iterable (like the words list) and joins the elements together with the string it's called on as a separator (in this case, a space " "). This approach is often preferred for its readability and performance, especially when dealing with many strings.

In summary, both methods achieve the same result, but the "".join() method is generally more efficient and Pythonic for string concatenation from a list.

**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.  
  



Explaination:

words = ["AI", "helps", "in", "refactoring", "code"] sentence = " ".join(words) print(sentence),tudent\_scores = {"Alice": 85, "Bob": 90} if "Charlie" in student\_scores: print(student\_scores["Charlie"]) else: print("Not Found") explain difference between these

The first snippet demonstrates two ways to join a list of words into a sentence. The first cell uses a for loop to iterate through the list and concatenate each word to the sentence variable, adding a space after each word. Finally, strip() is used to remove the trailing space. The second cell uses the join() method of strings, which is a more concise and Pythonic way to achieve the same result. It takes an iterable (like a list) as an argument and concatenates the elements with the string on which it's called as a separator.

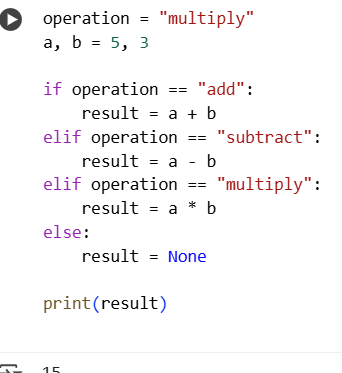
The third snippet shows two ways to check if a key exists in a dictionary and retrieve its value. The first cell uses an if-else statement and the in operator to check for the key's existence before attempting to access its value. If the key is not found, it prints "Not Found". The second cell uses the .get() method of dictionaries. This method allows you to specify a default value to return if the key is not found, which can make the code more compact and avoid potential KeyError exceptions. In this case, if "Charlie" is not found, it returns and prints "Not Found".

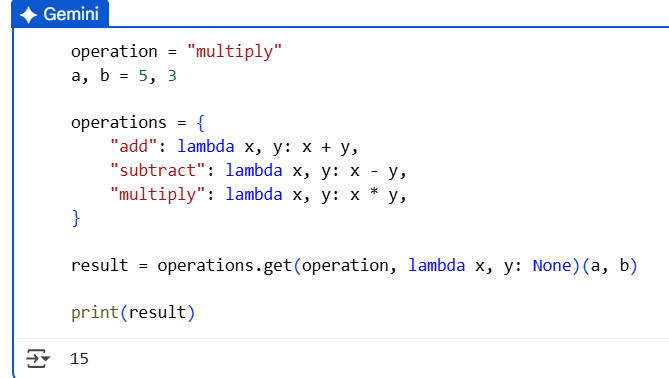
**Task 4**

**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.





Explaination:

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) ,peration = "multiply" a, b = 5, 3 operations = { "add": lambda x, y: x + y, "subtract": lambda x, y: x - y, "multiply": lambda x, y: x \* y, } result = operations.get(operation, lambda x, y: None)(a, b) print(result) explain difference between these

Both code snippets achieve the same result, which is to perform an arithmetic operation based on the value of the operation variable. However, they use different approaches:

The first snippet uses a series of if-elif-else statements. This is a straightforward way to handle multiple conditions, but it can become lengthy and less readable if you have many operations.

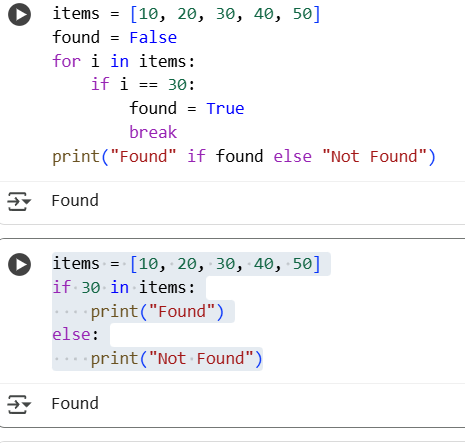
The second snippet uses a dictionary to map operation names (strings) to corresponding lambda functions (anonymous functions) that perform the calculations. This approach is often considered more concise and maintainable, especially when dealing with a larger number of possible operations. The .get() method of the dictionary is used to retrieve the appropriate function based on the operation variable, and it provides a default lambda function that returns None if the operation is not found in the dictionary.

**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.



Explaination:

Both of these code snippets check if the number 30 is present in the items list and print "Found" if it is, and "Not Found" otherwise.

The first snippet uses a for loop to iterate through each element in the items list. It uses a found flag, initialized to False. Inside the loop, it checks if the current element i is equal to 30. If it is, it sets found to True and immediately exits the loop using break. After the loop, it prints "Found" if found is True, and "Not Found" otherwise. This approach is a manual way of searching and can be useful for understanding the underlying process of iteration and conditional checking.

The second snippet uses the in keyword, which is a more Pythonic and efficient way to check for membership in a list (or other iterable). The expression 30 in items directly evaluates to True if 30 is found in the items list and False otherwise. This approach is generally preferred for its simplicity and readability, and it is often more optimized internally by Python than a manual loop for simple membership checks.