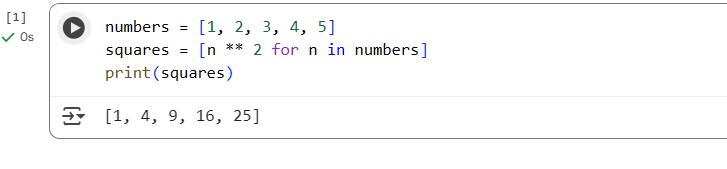
**ASSIGNMENT-13**

**NAME:** V.VIVEK VARDHAN

**HT NO:**2403A52097 **BATCH:** 24BTCAIAIB05

**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 AND OUTPUT:**

****

**EXPLANATION:**

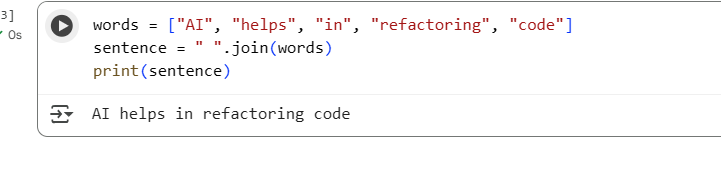
Certainly! Let's break down the code in cell 520075fa:

numbers = [1, 2, 3, 4, 5]  
squares = [n \*\* 2 for n in numbers]  
print(squares)

1. numbers = [1, 2, 3, 4, 5]: This line creates a list named numbers containing the integers from 1 to 5.
2. squares = [n \*\* 2 for n in numbers]: This is the core of the Pythonic approach, a **list comprehension**.
   * for n in numbers: This part iterates through each element in the numbers list. In each iteration, the current element is assigned to the variable n.
   * n \*\* 2: For each n, this part calculates its square (n raised to the power of 2).
   * [...]: The square brackets around the expression and the loop indicate that a new list is being created, where each element is the result of n \*\* 2 for each n in numbers.
3. print(squares): This line prints the contents of the newly created squares list to the console.

The result is the list [1, 4, 9, 16, 25], where each element is the square of the corresponding element in the numbers list. The list comprehension provides a concise and readable way to generate this new list based on an existing one.

**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 AND OUTPUT:  
**

**EXPLANATION:**

words = ["AI", "helps", "in", "refactoring", "code"]  
sentence = " ".join(words)  
print(sentence)

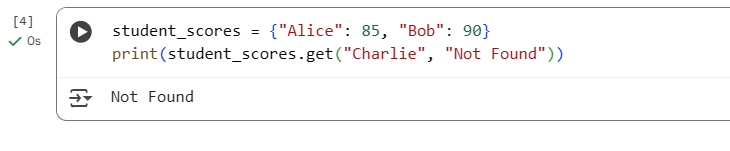
1. words = ["AI", "helps", "in", "refactoring", "code"]: This line creates a list named words containing several strings.
2. sentence = " ".join(words): This is the more Pythonic way to concatenate the strings in the words list.
   * .join() is a string method. You call it on the string that you want to use as a separator between the elements you are joining. In this case, " " is the separator, meaning a space.
   * words is the iterable (the list) whose elements will be joined together.
   * The join() method takes each string in the words list and concatenates them with a space in between each one.
3. print(sentence): This line prints the resulting concatenated string to the console.

The output is AI helps in refactoring code. Using "".join() is generally more efficient and readable than using a loop with += for string concatenation, especially with a large number of strings.

**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 AND OUPUT:**

****

**EXPLANATION:**

student\_scores = {"Alice": 85, "Bob": 90}  
print(student\_scores.get("Charlie", "Not Found"))

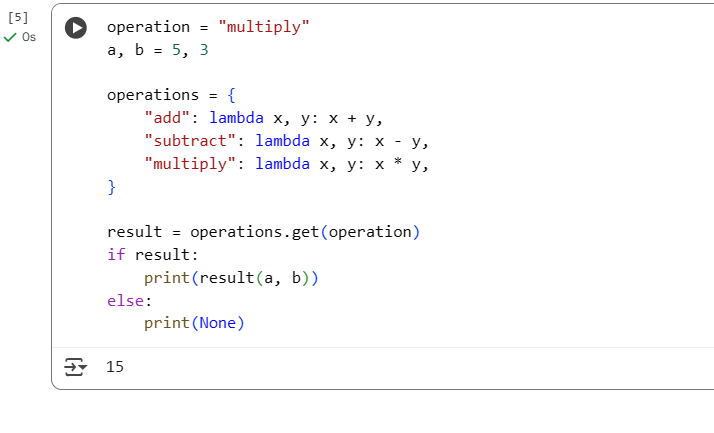
1. student\_scores = {"Alice": 85, "Bob": 90}: This line creates a dictionary called student\_scores where keys are student names (strings) and values are their scores (integers).
2. print(student\_scores.get("Charlie", "Not Found")): This is the key part that demonstrates a safer way to access dictionary values.
   * student\_scores.get("Charlie", "Not Found"): The .get() method is called on the student\_scores dictionary.
     + The first argument, "Charlie", is the key we are trying to look up in the dictionary.
     + The second argument, "Not Found", is the **default value**. This value is returned if the key ("Charlie" in this case) is **not found** in the dictionary.

In contrast to directly accessing a key using square brackets (e.g., student\_scores["Charlie"]), which would raise a KeyError if the key doesn't exist, .get() provides a way to specify what should be returned in that scenario. This makes the code more robust and prevents crashes when a key might be missing.

Since "Charlie" is not a key in the student\_scores dictionary, the .get() method returns the default value, "Not Found", which is then printed to the console.

**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 AND OUTPUT:**

****

**EXPLANATION:**

1. operation = "multiply" and a, b = 5, 3: These lines set the desired operation as a string and define the two numbers (a and b) that will be used in the operation.
2. operations = { ... }: This is the core of the refactoring. It creates a dictionary called operations.
   * The keys of the dictionary are strings representing the names of the operations ("add", "subtract", "multiply").
   * The values of the dictionary are **lambda functions**. A lambda function is a small, anonymous function.
     + lambda x, y: x + y is a lambda function that takes two arguments, x and y, and returns their sum (x + y).
     + Similarly, there are lambda functions for subtraction and multiplication.

This dictionary effectively maps the string name of an operation to the actual function that performs that operation.

1. result = operations.get(operation): This line uses the .get() method (which we discussed earlier) to look up the function associated with the operation string ("multiply") in the operations dictionary.
   * If the operation string is found as a key, the corresponding lambda function is returned and assigned to the result variable.
   * If the operation string is *not* found, .get() would return None by default (since we didn't provide a second argument like in the previous example), and None would be assigned to result.
2. if result:: This checks if the result variable is not None. Since operations.get("multiply") returns a function (which is not None), this condition is true.
3. print(result(a, b)): Because result holds the lambda function for multiplication, this line calls that function with a (5) and b (3) as arguments: (lambda x, y: x \* y)(5, 3), which evaluates to 5 \* 3 = 15. The value 15 is then printed.

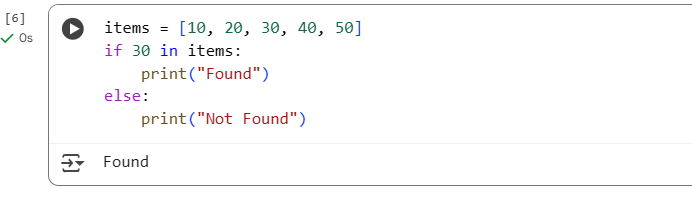
If operation was set to something like "divide", operations.get("divide") would return None, the if result: condition would be false, and the else block would execute, printing None.

This approach is more scalable because if you want to add a new operation (like "divide" or "power"), you just need to add a new entry to the operations dictionary with the operation name as the key and its corresponding lambda function as the value. You don't need to add another elif block.

**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 AND OUTPUT:**

****

**EXPLANATION:**

1. items = [10, 20, 30, 40, 50]: This line creates a list named items containing several integers.
2. if 30 in items:: This is the key part that shows the Pythonic way to check for membership in a list (or other iterable).
   * The in keyword is used to test if a sequence or a value is present in an object.
   * 30 is the value we are checking for.
   * items is the list we are searching within.
   * The expression 30 in items evaluates to True if the value 30 is found anywhere in the items list, and False otherwise.
3. print("Found"): If the condition 30 in items is True (which it is in this case), this line is executed, printing "Found".
4. else: print("Not Found"): If the condition 30 in items were False, this else block would be executed, printing "Not Found".

This approach is much simpler and more readable than the original code that used a for loop and a boolean flag (found). Python's in keyword is specifically designed for membership testing and is generally optimized for this task.