ASSIGNMENT

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 given by google colab :

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

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

Explanation of the code:

1. `numbers = [1, 2, 3, 4, 5]`: This line initializes a list named `numbers` with integer values from 1 to 5. This is the input data for our calculation.
2. `squares = [n ** 2 for n in numbers]`: This is the refactored part, using a **list comprehension**. It's doing the same job as the original `for` loop but in a more Pythonic way.
    - `[ ]`: Creates a new list.
    - `n ** 2`: This is the operation performed on each item. It calculates the square of the current number (`n`).
    - `for n in numbers`: This iterates through each item (`n`) in the `numbers` list.

   So, for each number in `numbers`, the code calculates its square and adds it to the new list called `squares`.
3. `print(squares)`: This line prints the contents of the `squares` list to the console. As expected, this output is `[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 GIVEN BY THE GOOGLE COLAB :

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

```
AI helps in refactoring code
```

EXPLANATION OF THE CODE :

1. `words = ["AI", "helps", "in", "refactoring", "code"]`: This line creates a list of strings called `words`. Each element in the list is a single word.
2. `sentence = " ".join(words)`: This is the key part where the refactoring happens.
   - `" "`: This is the string that will be used as a separator between the elements of the list. In this case, it's a space, so the words will be joined with a space in between them.
   - `.join(words)`: This is a string method that takes an iterable (like our `words` list) and concatenates its elements into a single string. The elements of the iterable are joined by the string on which the `join()` method is called.

   This line effectively takes the words in the `words` list and combines them into a single sentence string, with each word separated by a space.
3. `print(sentence)`: This line prints the resulting `sentence` string to the output.

This approach using `"".join()` is more efficient and Pythonic for joining strings from a list compared to repeatedly using the `+=` operator in a loop, especially for large lists.

Does that explanation help? Let me know if you'd like me to explain anything else!

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")

Expected Output:
Not Found

CODE GIVEN BY GOOGLE COLAB :

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

```
Not Found
```

EXPLANATION OF THE CODE:

1. `student_scores = {"Alice": 85, "Bob": 90}`: This line initializes a dictionary named `student_scores`. This dictionary stores student names as keys and their corresponding scores as values.
2. `score = student_scores.get("Charlie", "Not Found")`: This is the refactored part using the `.get()` method.
   - `student_scores.get()`: This method is used to access the value associated with a key in a dictionary.
   - `"Charlie"`: This is the key we are trying to look up in the `student_scores` dictionary.
   - `"Not Found"`: This is the **default value**. If the key `"Charlie"` is found in the dictionary, the method will return the corresponding value (which would be the score). However, if the key `"Charlie"` is *not* found (as is the case here), the method will return this default value `"Not Found"` instead of raising a `KeyError`.

   This line attempts to get the score for "Charlie". Since "Charlie" is not a key in the `student_scores` dictionary, the `.get()` method returns the default value "Not Found", and this value is assigned to the variable `score`.
3. `print(score)`: This line prints the value stored in the `score` variable to the output. In this case, it will print "Not Found".

Using `.get()` with a default value is a "safer" and more "Pythonic" way to access dictionary values because it prevents the program from crashing with a `KeyError` if the key doesn't exist.

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)
```

Expected Output:
15

CODE GIVEN BY GOOGLE COLAB :

```python
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
}

result = operations.get(operation)
if result:
    print(result(a, b))
else:
    print(None)
```

```
15
```

EXPLANATION OF THE CODE:

1. **`operation = "multiply"`**: This line sets a variable `operation` to the string "multiply". This string will be used to determine which mathematical operation to perform.
2. **`a, b = 5, 3`**: This line initializes two variables, `a` and `b`, with the values 5 and 3 respectively. These are the numbers that will be used in the operation.
3. **`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. Each lambda function takes two arguments (`x` and `y`) and performs a specific mathematical operation (`x + y`, `x - y`, or `x * y`).

   This dictionary effectively maps the name of an operation (a string) to the actual code that performs that operation (a function).
4. **`result = operations.get(operation)`**: This line uses the `.get()` method on the `operations` dictionary.
    - It tries to find the value associated with the key stored in the `operation` variable (which is "multiply").
    - Since "multiply" is a key in the `operations` dictionary, the corresponding value (the

5. **`if result:`**: This line checks if the `result` variable is not `None`. In this case, since `operations.get("multiply")` returned a function (which is not `None`), the condition is true.
6. **`print(result(a, b))`**: Since the `if` condition is true, this line is executed.
    - `result` currently holds the lambda function for multiplication.
    - `result(a, b)` calls this function with `a` (5) and `b` (3) as arguments. The function performs the multiplication (5 * 3 = 15).
    - `print()` then prints the result of the function call (15) to the output.
7. **`else: print(None)`**: This `else` block would only be executed if `operations.get(operation)` had returned `None` (which would happen if the value of `operation` was not a key in the `operations` dictionary).

This approach is cleaner and more scalable than the original if-elif-else chain because you can easily add new operations by simply adding a new key-value pair to the `operations` dictionary, without needing to add more `elif` blocks.

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 GIVEN BY GOOGLE COLAB :

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

```
Found
```

EXPLANATION OF THE CODE :

1. `items = [10, 20, 30, 40, 50]`: This line initializes a list named `items` containing several integer values. This is the list we want to search within.
2. `found = 30 in items`: This is the refactored part, using the `in` keyword.
   - The `in` keyword is a Python operator that checks for membership.
   - `30 in items` evaluates whether the value `30` exists as an element within the `items` list.
   - This expression returns a boolean value: `True` if `30` is found in the list, and `False` otherwise.
   - The result of this boolean check is assigned to the variable `found`.

   This single line replaces the original `for` loop and `if` condition, making the code much more concise and readable.
3. `print("Found" if found else "Not Found")`: This line uses a conditional expression (sometimes called a ternary operator) to print the result.
   - `"Found" if found else "Not Found"` is a compact way of writing an if-else statement.
   - If the `found` variable is `True`, the expression evaluates to the string `"Found"`.
   - If the `found` variable is `False`, the expression evaluates to the string `"Not Found"`.
   - The `print()` function then outputs the resulting string to the console.

This refactored code is more efficient because Python's `in` operator is highly optimized for checking membership in lists and other iterable objects.