AI-ASSISSTANT-CODING-LAB-13.4

Name: V.Abhinav

Batch:41

Roll-No:2303A52174

Task 1: Refactoring Data Transformation Logic

Task Description:

Review the legacy code that computes transformed values using an explicit loop. Use an AI tool to suggest a more Pythonic refactoring approach. Refactor the code using list comprehensions or helper functions while preserving the output.

Prompt:

How can I refactor this legacy Python code to be more Pythonic and efficient using a list comprehension instead of a multi-line for-loop? legacy code: values = [2, 4, 6, 8, 10]; doubled = []; for v in values: doubled.append(v * 2); print(doubled)

Code:

values = [2, 4, 6, 8, 10]

# Refactored: Using a list comprehension to create the new list in a single, readable line
doubled = [v * 2 for v in values]

print(doubled)

Output:

[4, 8, 12, 16, 20]

Justification:

I refactored the explicit for loop and the .append() method into a list comprehension. List comprehensions are highly considered the "Pythonic" way to transform lists. They are much easier to read at a glance because the intent is consolidated into a single line. Furthermore,

they are often faster in terms of execution time because the iteration happens internally within Python's underlying C implementation rather than making repeated append calls in the Python interpreter.

Task 2: Improving Text Processing Code Readability

Task Description:

Analyze the legacy code that constructs a sentence using repeated string concatenation. Ask AI to suggest a more efficient and readable approach. Refactor the code accordingly while keeping the final output unchanged.

Prompt:

Suggest a more efficient and readable approach to concatenate strings from a list in Python, avoiding the += operator inside a loop. legacy code: words = ["Refactoring", "with", "AI", "improves", "quality"]; message = ""; for w in words: message += w + " "; print(message.strip())

Code:

```
words = ["Refactoring", "with", "AI", "improves", "quality"]

# Refactored: Using the string .join() method for efficient array-to-string concatenation
message = " ".join(words)

print(message)
```

Output:

```
Refactoring with AI improves quality
```

Justification:

In Python, strings are immutable, meaning they cannot be changed once created. The legacy code was using += inside a loop, which is a bad practice because it forces Python to create a brand new string in memory during every single iteration, leading to poor performance and unnecessary memory allocation. I refactored this using the .join() method, which is specifically optimized to concatenate an iterable of strings in one single pass. It also automatically handles the spacing without needing the .strip() cleanup at the end.

Task 3: Safer Access to Configuration Data

Task Description:

Review the legacy code that manually checks for dictionary keys. Use AI suggestions to refactor the code using safer dictionary access methods. Ensure the behavior remains the same for missing keys.

Prompt:

Refactor this Python code to use a safer and more concise dictionary access method instead of manually checking for keys with an if-else block. legacy code: config = {"host": "localhost", "port": 8080}; if "timeout" in config: print(config["timeout"]) else: print("Default timeout used")

Code:

```
config = {"host": "localhost", "port": 8080}

# Refactored: Using dict.get() which safely attempts to fetch a key and returns a fallback default if missing
print(config.get("timeout", "Default timeout used"))
```

Output:

```
Default timeout used
```

Justification:

The original code used an if-else block to explicitly check if the key existed to avoid a KeyError. I refactored this using the built-in dictionary method .get(). This method takes two arguments: the key you want to find, and an optional default value to return if the key doesn't exist. This condenses four lines of branching logic down into a single, clean line while perfectly preserving the fallback behavior. It is much safer and represents standard Python best practices.

Task 4: Refactoring Conditional Logic for Scalability

Task Description:

Examine the multiple if–elif conditions used to determine operations. Ask AI to suggest a cleaner, scalable alternative. Refactor the logic using mapping techniques while preserving functionality.

Prompt:

I have a long if-elif block mapping strings to math operations. How can I refactor this using mapping techniques (like a dictionary) to make it cleaner and more scalable? legacy code: action = "divide"; x, y = 10, 2; if action == "add": result = x+y; elif action == "subtract": ... [rest of conditions]

Code:

```
action = "divide"
x, y = 10, 2

# Refactored: Using a dictionary to map string actions to lambda functions (Dispatch Table pattern)
operations = {
   "add": lambda a, b: a + b,
   "subtract": lambda a, b: a - b,
   "multiply": lambda a, b: a * b,
   "divide": lambda a, b: a / b
}

# .get() fetches the correct lambda function, and we immediately execute it by passing (x, y)
# The default fallback is a lambda that simply returns None
result = operations.get(action, lambda a, b: None)(x, y)

print(result)
```

Output:

```
5.0
```

Justification:

Long if-elif chains are notoriously hard to maintain; if we wanted to add 10 more mathematical operations, the code would grow unnecessarily long. I refactored this using a "Dispatch Table" approach. By mapping the action strings to executable lambda functions inside a dictionary, we separate our data from our control flow logic. We can now evaluate the correct operation in O(1) lookup time. To execute it, we use .get() to pull the right function and pass the variables to it immediately, resulting in highly scalable and maintainable code.

Task 5: Simplifying Search Logic in Collections

Task Description:

Identify the explicit loop used for searching an item. Use AI assistance to refactor the logic into a more concise and readable form. Maintain the same output behavior.

Prompt:

Refactor this list searching logic. How can I avoid using an explicit for-loop and a boolean flag variable just to check if an item exists in a list? legacy code: inventory = ["pen", "notebook", "eraser", "marker"]; found = False; for item in inventory: if item == "eraser": found = True; break; print("Item Available" if found else "Item Not Available")

Code:

```
inventory = ["pen", "notebook", "eraser", "marker"]

# Refactored: Using the 'in' operator to implicitly check for membership and avoiding manual loops
is_available = "eraser" in inventory

print("Item Available" if is_available else "Item Not Available")
```

Output:

```
Item Available
```

Justification:

The legacy code used a manual loop, a boolean flag variable (found), and a break statement to stop searching once the item was found. This is a common pattern in languages like C or Java, but it is unnecessary in Python. I refactored it using Python's built-in in operator. The in operator automatically handles the iteration and the early exit (the break equivalent) under the hood. This reduces the code drastically, removes the need for state-tracking variables, and expresses the intent (checking for membership) in plain English syntax.