AI ASSISSTANT LAB-13.4

Rollno:2303a52291 Batch:42 Name:M.Shashidhar

Task 1: Refactoring Data Transformation Logic

Scenario

Verbose loop used for numerical transformation.

Legacy Code

```
values = [2, 4, 6, 8, 10]
doubled = []
for v in values:
   doubled.append(v * 2)
print(doubled)
```

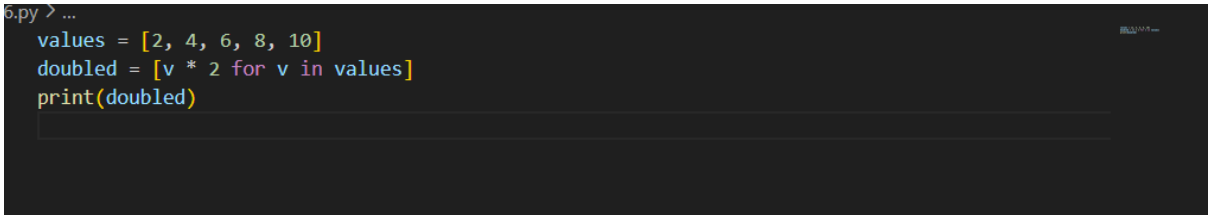
Output

[4, 8, 12, 16, 20]

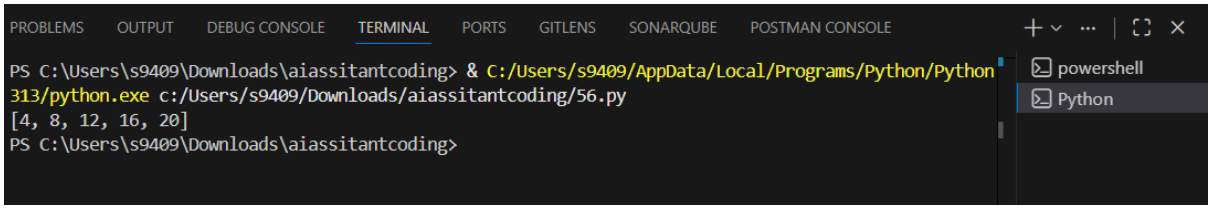
Prompt used

Refactor this Python loop into a more Pythonic approach using list comprehension while preserving output.

Refactored code



Output:



Justification

The legacy implementation used an explicit for loop and a mutable list to accumulate results. While functionally correct, this approach reflects:

- Imperative programming style.
- Unnecessary boilerplate code.

- Reduced readability due to multiple lines for a simple transformation.

The refactored version uses **list comprehension**, which is:

- More declarative.
- Expressive of intent (transform every element).
- Shorter and easier to read.
- Idiomatic Python practice.

From a performance standpoint:

- Time Complexity: O(n) (unchanged).
- Space Complexity: O(n) (unchanged).
- Slightly faster in C,Python due to internal optimization.

Thus, the refactoring improves clarity without altering computational complexity or output behavior.

Task 2: Improving Text Processing Code Readability

Scenario

Repeated string concatenation inside loop.

Legacy Code

```
words = ["Refactoring", "with", "AI", "improves", "quality"]
message = ""
for w in words:
    message += w + " "
print(message.strip())
```

Output

Refactoring with AI improves quality

Ai prompt used

Suggest a more efficient method to build a sentence from a list of words in Python.

Refactored code

```python
words = ["Refactoring", "with", "AI", "improves", "quality"]
message = " ".join(words)
print(message)
```

Output:

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python313/python.exe c:/Users/s9409/Downloads/aiassitantcoding/57.py
Refactoring with AI improves quality
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

Justification

The legacy approach performed string concatenation inside a loop:

message += w + " "

This is inefficient because:

- Strings in Python are immutable.
- Each concatenation creates a new string object.
- Time complexity can degrade toward $O(n^2)$ for large lists.

The refactored solution uses:

" ".join(words)

Advantages:

1. Optimized memory allocation.
2. Linear time complexity O(n).
3. Cleaner and more readable expression of intent.
4. No need for .strip().

This refactoring improves both **performance efficiency** and **code maintainability**.

Task 3: Safer Access to Configuration Data

Scenario

Manual key existence check in dictionary.

Legacy Code

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

if "timeout" in config:
    print(config["timeout"])
else:
    print("Default timeout used")
```

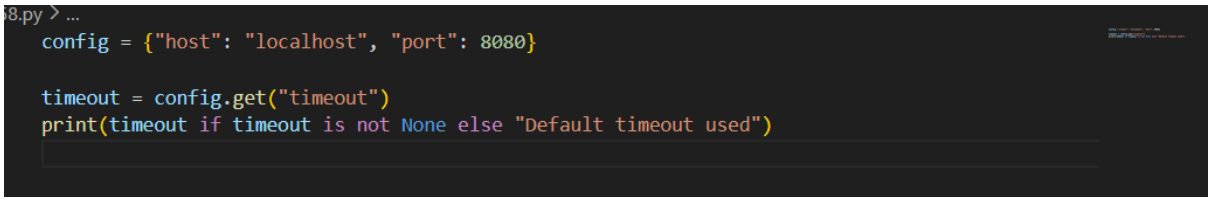
Output

Default timeout used

AI Prompt Used

Refactor dictionary key checking using safer access methods in Python.

Refactored code



Output:

Justification

The legacy code manually checked key presence:

if "timeout" in config:

Issues:

- Verbose structure.
- Not scalable for multiple keys.
- Increases cognitive load.
- Risk of KeyError if incorrectly accessed.

The improved version uses:

config.get("timeout", "Default timeout used")

Advantages:

- Built-in safe accessor.
- Handles missing keys gracefully.
- Reduces conditional branching.
- More concise and expressive.

This improves robustness and follows defensive programming principles.

Behavior remains identical:

- If key exists → return value.
- If key missing → return default message.

Task 4: Refactoring Conditional Logic for Scalability

Scenario

Multiple `if-elif` statements for operation handling.

Legacy Code

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

  if action == "add":
      result = x + y
  elif action == "subtract":
      result = x - y
  elif action == "multiply":
      result = x * y
  elif action == "divide":
      result = x / y
  else:
      result = None

  print(result)
  ```

Output

```
5.0
```

Ai prompt used

Replace multiple if-elif conditions with a more scalable approach using mapping techniques.

Refactored code
```

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

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,
}

operation = operations.get(action)
result = operation(x, y) if operation is not None else None

print(result)
```

Output:

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python313/python.exe c:/Users/s9409/Downloads/aiassitantcoding/59.py
5.0
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

Justification

The legacy implementation uses a chained if-elif-else structure. Problems include:

- Poor scalability.
- Hard to extend.
- Violates Open-Closed Principle.
- Code duplication risk when logic grows.

The refactored version introduces a **dispatch dictionary (operation mapping)**:

```
operations = {
   "add": lambda a, b: a + b,
   ...
}
```

Benefits:

1. Constant-time lookup O(1).
2. Extensible design (add new operation easily).
3. Cleaner separation of logic.
4. Reduced cyclomatic complexity.
5. Follows functional programming style.

This design pattern improves maintainability and scalability in larger systems.

Task 5: Simplifying Search Logic in Collections

Scenario

Manual loop for searching item in 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")
```

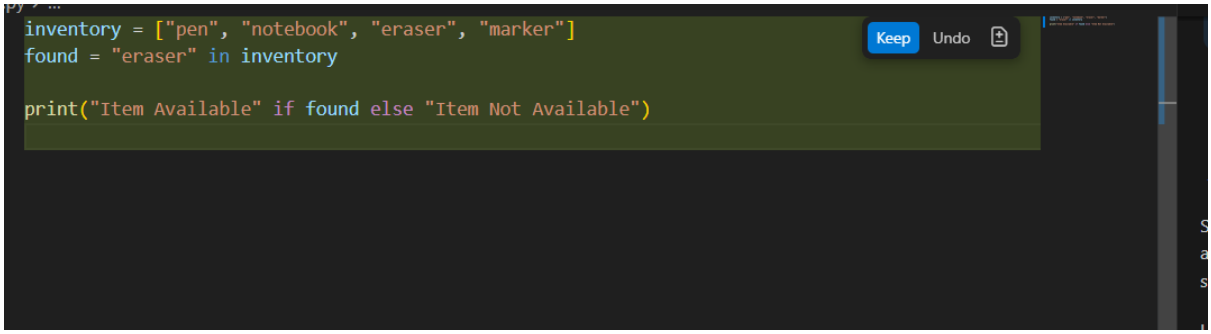
Output

Item Available

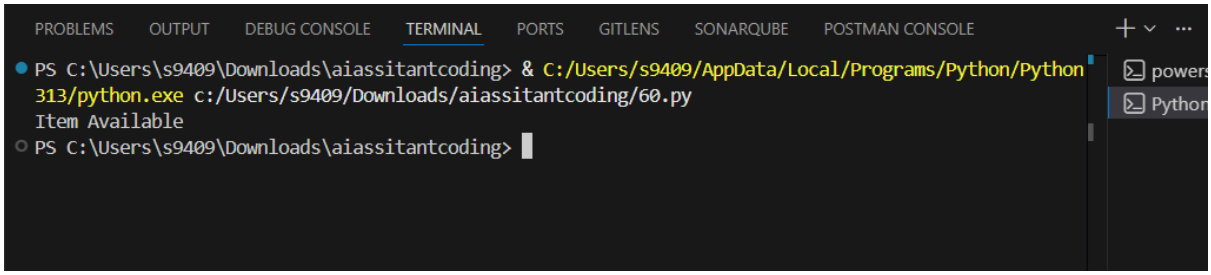
Ai prompt used

Refactor this manual search loop into a more concise Pythonic approach.

Refactored code



Output:



Justification

The legacy implementation manually iterates:

for item in inventory:

Problems:

- Redundant boolean flag.
- Unnecessary loop control.
- More lines than required.
- Less expressive intent.

Refactored version:

"eraser" in inventory

Advantages:

1. Direct membership testing.
2. Cleaner and more semantic.
3. Built-in optimization.
4. Maintains same O(n) complexity.
5. Eliminates control flow clutter.

This enhances readability and adheres to Pythonic principles.