ASSIGNMENT-13

2403A52153-PALLAPUVISHNU

TASK1:

PROMPT

Write a python program that Refactor the following legacy code to use a cleaner, more Pythonic list comprehension while producing the same output.

CODE:

```
numbers = [1, 2, 3, 4, 5]

squares = [n ** 2 for n in numbers]

print(squares)
```

OUTPUT

[1, 4, 9, 16, 25]

EXPLANATION

The list comprehension [n ** 2 for n in numbers] is the Pythonic replacement for the explicit for loop and append operation.

1. for n in numbers: This part iterates over each element in the numbers list, assigning the value of each element to the temporary variable n.

2. n ** 2: This is the expression evaluated for each item. It calculates the square of the current value of n.

3. [...]: The square brackets indicate that the result of this operation should be collected into a new list.

This single line achieves the same result as the three lines of the legacy code (squares = [], the for loop header, and squares.append(...)), making the code more concise and readable.

TASK-2:

PROMPT:

Refactor the following Python code, which uses a loop for string concatenation, into a single, more efficient line using the " ".join() method. Keep the output identical to the expected result.

CODE:

```
words = ["AI", "helps", "in", "refactoring", "code"]

sentence = ""

for word in words:

  sentence += word + " "

print(sentence.strip())

sentence_refactored = " ".join(words)

print(sentence_refactored)
```

OUTPUT:

AI helps in refactoring code

AI helps in refactoring code

EXPLANATION:

The " ".join(words) method is a single, highly optimized command that replaces the entire for loop, the empty string initialization, and the repeated string concatenation.

1. Efficiency: In the legacy code, the line sentence += word + " " creates a brand-new string object in memory in every single iteration of the loop. This repeated creation and copying is inefficient. The str.join() method, on the other hand, calculates the final required string size once and builds the complete string directly in memory, which is significantly faster, especially for long lists of words.

2. Readability: The code is reduced to one line and clearly states the intent: "Join these words using a space (" ") as the delimiter."

3. Handling Spaces: The legacy code required a final call to sentence.strip() to remove the extra space added after the last word. The str.join() method automatically handles this by only inserting the separator between the elements, resulting in a clean sentence without needing the extra strip() call.

TASK-3:

PROMPT:

Refactor the following legacy code, which uses an explicit if/else block to check for a dictionary key, to instead use the dict.get() method for a safer and more concise lookup. The default value should be "Not Found."

CODE:

```
student_scores = {"Alice": 85, "Bob": 90}

if "Charlie" in student_scores:

  print(student_scores["Charlie"])

else:

  print("Not Found")

print(student_scores.get("Charlie", "Not Found"))
```

OUTPUT

Not Found

Not Found

EXPLANATION

The dict.get(key, default_value) method is the preferred Pythonic way to retrieve a dictionary value when the key's existence is uncertain.

1. Direct Retrieval: The legacy code required four lines (if, in, print with bracket access, and else) to handle the missing key scenario.

2. Conciseness and Safety: The refactored code does this in a single, safe line. It attempts to look up the key "Charlie":

o If the key is found, it returns the associated value (e.g., 85).

o If the key is not found, it returns the specified default value, which is "Not Found".

3. Avoiding Errors: Using the unsafe method, student_scores["Charlie"], would raise a KeyError if the key didn't exist, crashing the program. The .get() method prevents this error gracefully.

TASK-4:

PROMPT:

Refactor the following legacy code, which uses multiple if-elif statements to perform arithmetic operations, into a cleaner, more scalable Pythonic approach using a dictionary to map operation names to corresponding functions or lambda expressions.OMPT

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)
operations = {
   "add": lambda x, y: x + y,
   "subtract": lambda x, y: x - y,
   "multiply": lambda x, y: x * y,
}

result_refactored = operations.get(operation, lambda x, y: None)(a, b)

print(result_refactored)
```

OUTPUT:

15

15

EXPLANATION:

This refactoring replaces the sprawling if-elif-else block with a single dictionary lookup, dramatically improving code cleanliness and scalability (adding a new operation only requires adding one entry to the dictionary, not a new elif block).

1. operation_map Dictionary: This dictionary stores the logic. Each key is the string identifier (e.g., "multiply"), and the corresponding value is a lambda function that performs the actual math (e.g., lambda x, y: x * y).

2. dict.get(operation, default_func): This is the core of the refactoring.

o It retrieves the function associated with the operation string ("multiply").

o The second argument, lambda x, y: None, acts as a safe default for the else: block in the original code. If a non-existent operation is requested, func will be set to this default function, ensuring a safe return of None.

3. result = func(a, b): Finally, the retrieved function (func) is executed by passing it the variables a (5) and b (3), which calculates the result (15).

TASK-5:

PROMPT:

Refactor the following legacy code, which uses an explicit for loop and a break statement to check for the presence of an item in a list, to use the more idiomatic and efficient in keyword for the search operation.

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")
print("Found" if 30 in items else "Not Found")
```

OUTPUT:

Found

Found

EXPLANATION:

The in keyword is the standard and most efficient way to check for element membership in any Python iterable (like lists, tuples, or sets).

1. Efficiency and Readability: The legacy code manually simulates a search by looping, checking, and setting a flag (found = True) before using break. The refactored code replaces this entire four-line block with the concise and readable expression: found = target in items.

2. Boolean Result: The expression target in items directly evaluates to a boolean value (True if the item is found, False otherwise), which can be directly assigned to the found variable.

3. Simplicity: This method is not only cleaner but is also often optimized internally by Python's interpreter, making it as fast or faster than a manually coded loop.

4. Conditional Print: The final print("Found" if found else "Not Found") line remains a good Pythonic practice, cleanly condensing the final if/else logic into a single expression.