**ASSIGNMENT – 13.4**

**NAME :** SAI CHARAN .P

**ROLL:NO :** 2403A52343

**BATCH :** AI 13

**TASK-1:** You are given legacy Python code that computes the square of each number in a list using a for loop and append. Your task is to refactor the loop into a more concise and Pythonic form using list comprehension, while ensuring the output remains exactly the same.

**Legacy Code:**

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

squares = []

for n in numbers:

squares.append(n \*\* 2)

print(squares)

**Refactored Code:**

A number and a phone number

AI-generated content may be incorrect.

**Output:**



**Explanation:**

The code does the following:

1. numbers = [1, 2, 3, 4, 5]: This line creates a list named numbers and initializes it with the integers from 1 to 5.
2. squares = [n \*\* 2 for n in numbers]: This is a list comprehension. It's a concise way to create a new list.
   * It iterates through each element (n) in the numbers list.
   * For each n, it calculates its square (n \*\* 2).
   * It collects all these calculated squares into a new list called squares.
3. print(squares): This line prints the contents of the squares list to the console.

The output [1, 4, 9, 16, 25] is the result of printing the squares list, which contains the squares of the numbers 1 through 5.

**TASK-2:** You are given legacy code that builds a sentence by concatenating words in a list using a for loop and the += operator. Your task is to refactor the code to use " ".join(), which is more efficient and Pythonic, while maintaining the exact same output.

**Legacy Code:**

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

sentence = ""

for word in words:

sentence += word + " "

print(sentence.strip())

**Refactored Code:**

A close up of words

AI-generated content may be incorrect.

**Output:**



**Explanation:**

The code does the following:

1. words = ["AI", "helps", "in", "refactoring", "code"]: This line creates a list named words containing several strings.
2. sentence = " ".join(words): This is the key part. The join() method is a string method that concatenates the elements of an iterable (like our words list) into a single string.
   * The string before the .join() method (" ") is the separator that will be placed *between* each element from the list. In this case, it's a single space.
   * So, it takes each word from the words list and joins them together with a space in between.
3. print(sentence): This line prints the resulting sentence string to the console.

The output AI helps in refactoring code is the string created by joining the elements of the words list with spaces.

**TASK-3:** You are given legacy code that checks if a key exists in a dictionary using an if...else statement before accessing the value. Your task is to refactor the code using the .get() method to make it more concise and Pythonic, while ensuring the output remains the same.

**Legacy Code:**

student\_scores = {"Alice": 85, "Bob": 90}

if "Charlie" in student\_scores:

print(student\_scores["Charlie"])

else:

print("Not Found")

**Refactored Code:**

A close up of text

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

**Output:**



**Explanation:**

The code does the following:

1. student\_scores = {"Alice": 85, "Bob": 90}: This line creates a dictionary called student\_scores. This dictionary stores student names as keys and their corresponding scores as values.
2. score = student\_scores.get("Charlie", "Not Found"): This line uses the .get() method to retrieve a value from the dictionary.
   * "Charlie" is the key we are trying to look up in the student\_scores dictionary.
   * "Not Found" is the default value. If the key "Charlie" is *not* found in the dictionary, the .get() method will return this default value instead of raising a KeyError.
   * The result of the .get() call is assigned to the variable score.
3. print(score): This line prints the value stored in the score variable.

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

**TASK-4:** You are given legacy code that performs different arithmetic operations using a chain of if-elif statements. Your task is to refactor this into a dictionary-based approach, making it more scalable and readable, while keeping the output the same.

**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)

**Refactored Code:**

A black and white text on a white background

AI-generated content may be incorrect.

**Output:**



**Explanation:**

The code does the following:

1. operation = "multiply": This line assigns the string "multiply" to the variable operation. This variable determines which mathematical operation will be performed.
2. a, b = 5, 3: This line uses tuple unpacking to assign the value 5 to variable a and 3 to variable b. These are the operands for the operation.
3. operations = {...}: This creates a dictionary called operations.
   * The keys of this dictionary are strings representing the names of the operations ("add", "subtract", "multiply").
   * The values are lambda functions. A lambda function is a small, anonymous function. Each lambda function takes two arguments (x and y) and performs the corresponding mathematical operation (x + y, x - y, or x \* y).
4. result = operations.get(operation)(a, b) if operation in operations else None: This is a conditional expression (also known as a ternary operator) combined with a dictionary lookup and function call. Let's break it down:
   * if operation in operations: This checks if the value of the operation variable ("multiply") exists as a key in the operations dictionary.
   * operations.get(operation): If the operation key is found in the dictionary, this part retrieves the corresponding value, which is a lambda function. Using .get() is safer than direct access (operations[operation]) because it returns None if the key isn't found, preventing a KeyError.
   * (a, b): This immediately calls the retrieved lambda function (the one for "multiply" in this case) with a (5) and b (3) as arguments.
   * operations.get(operation)(a, b): So, if the operation is found, this whole part evaluates to the result of calling the function with a and b.
   * else None: If the operation key is *not* found in the operations dictionary, the entire expression evaluates to None.
   * result = ...: The final evaluated value is assigned to the result variable.
5. print(result): This line prints the value of the result variable.

Since operation is "multiply" and "multiply" is a key in the operations dictionary, the lambda function for multiplication (lambda x, y: x \* y) is retrieved and called with a=5 and b=3. The result 5 \* 3 = 15 is assigned to result and then printed.

**TASK-5:** You are given legacy code that uses a loop to search for a specific element in a list and sets a flag if it is found. Your task is to refactor the code using Python’s in keyword, which is more efficient and concise, while preserving the exact same output.

**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")

**Refactored Code:**

A close-up of a computer code

AI-generated content may be incorrect.

**Output:**



**Explanation:**

The code does the following:

1. items = [10, 20, 30, 40, 50]: This line creates a list named items containing five integer values.
2. if 30 in items:: This is the core of the code. It uses the in operator to check for membership. It asks: "Is the value 30 present within the items list?"
3. print("Found"): If the condition in the if statement is true (meaning 30 is found in the items list), this line prints the string "Found".
4. else:: This keyword introduces the block of code to be executed if the if condition is false.
5. print("Not Found"): If the condition in the if statement is false (meaning 30 is not found in the items list), this line prints the string "Not Found".

In this case, since the value 30 is indeed present in the items list, the if condition is true, and the output is "Found". Using the in operator is the most Pythonic and efficient way to check if an element exists within a list or other iterable.