AI ASSISTED CODING

ASSIGNMENT 13.4

HALL TICKNO:2403A52376

BATCH :14AI

**TASK - 1**

CODE:

A black background with white text and numbers

AI-generated content may be incorrect.

OUTPUT:



EXPLANATION :

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, calculates the square of that element (n \*\* 2), and adds the result to 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] shows the squares of the numbers 1 through 5.

TASK -2

CODE:

A black background with white text

AI-generated content may be incorrect.

OUTPUT:



CODE EXPLANATION :

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 for string concatenation. The join() method is called on a string (in this case, a space " "). It takes an iterable (like our words list) as an argument and concatenates the elements of the iterable into a single string, using the string it was called on as a separator between the elements. 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 single string formed by joining the words in the list with spaces. This method is generally more efficient than repeatedly using += for string concatenation in a loop, especially for larger lists.

**TASK -3**

**CODE:**

**A black background with white text

AI-generated content may be incorrect.**

**OUPUT:**

****

**CODE EXPLANATION :**

1. **student\_scores = {"Alice": 85, "Bob": 90}: This line creates a dictionary named student\_scores. Dictionaries store key-value pairs. In this case, the keys are student names (strings) and the values are their scores (integers).**
2. **print(student\_scores.get("Charlie", "Not Found")): This line uses the .get() method to access a value in the dictionary.**
   * **student\_scores.get("Charlie", ...): This attempts to retrieve the value associated with the key "Charlie".**
   * **The second argument, "Not Found", is the *default value*. If the key "Charlie" is found in the dictionary, its corresponding value is returned. If the key is *not* found, the .get() method returns this default value instead of raising a KeyError.**
   * **The result of the .get() method (either the value for "Charlie" or the default "Not Found") is then printed to the console.**

**In this specific case, "Charlie" is not a key in the student\_scores dictionary, so the .get() method returns the default value "Not Found", which is then printed. This is a safer way to access dictionary values when you're not sure if a key exists, as it avoids potential errors.**

**TASK-4**

**CODE:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**OUTPUT:**

****

**CODE EXPLANATION:**

1. **operation = "multiply": This line sets a variable operation to the string "multiply". This string will determine which operation is performed.**
2. **a, b = 5, 3: This line assigns the values 5 and 3 to variables a and b respectively. These are the numbers that will be used in the chosen operation.**
3. **operations = { ... }: This creates a dictionary named operations. The keys of this dictionary are the names of the operations (strings like "add", "subtract", "multiply"), and the values are anonymous functions (defined using lambda) that perform the actual calculations.**
   * **"add": lambda x, y: x + y: This maps the string "add" to a function that takes two arguments (x and y) and returns their sum.**
   * **"subtract": lambda x, y: x - y: This maps "subtract" to a function that returns the difference between x and y.**
   * **"multiply": lambda x, y: x \* y: This maps "multiply" to a function that returns the product of x and y.**
4. **result = operations.get(operation, lambda x, y: None)(a, b): This is the core of the refactored code.**
   * **operations.get(operation, lambda x, y: None): This uses the .get() method on the operations dictionary to retrieve the function associated with the value of the operation variable (which is "multiply"). The second argument to .get() is a default value – in this case, another lambda function that returns None. This ensures that if the operation string doesn't match any key in the dictionary, a default function (which does nothing and returns None) is returned instead of raising a KeyError.**
   * **(a, b): The result of the .get() call is a function (the lambda function for multiplication in this case). We immediately call this retrieved function with the arguments a and b.**
5. **print(result): This line prints the value returned by the called function, which is the result of the operation.**

**TASK -5**

**CODE:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**OUTPUT :**

****

**CODE EXPLANATION :**

1. **items = [10, 20, 30, 40, 50]: This line creates a list named items and initializes it with a few integer values.**
2. **if 30 in items:: This is the core of the search. The in keyword is used to check for membership. It evaluates to True if the element 30 is found anywhere within the items list, and False otherwise.**
3. **print("Found"): This line is executed if the condition in the if statement is True (meaning 30 was found in the list).**
4. **else: print("Not Found"): This block is executed if the condition in the if statement is False (meaning 30 was not found in the list).**