

## AI ASSISTED CODING

### LAB ASSIGNMENT – 13.4

HALL TICKET NO : 2403A52394


BATCH NO : 14

Task – 01:

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

```
squares = [n ** 2 for n in numbers] print(squares)
```

OUTPUT :

 `[1, 4, 9, 16, 25]`

### EXPLANATION :

This code snippet 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.

So, the code calculates the square of each number in the numbers list and stores them in the squares list, which is then printed.

TASK – 02:

```
words = ["AI", "helps", "in", "refactoring", "code"] sentence = " ".join(words)
print(sentence) OUTPUT 01:
```

```
➞ AI helps in refactoring code
```

```
numbers = [1, 2, 3, 4, 5]
squares = [n ** 2 for n in numbers] print(squares)
```

OUTPUT 02:

```
➞ [1, 4, 9, 16, 25]
```

EXPLANATION :

This code snippet performs the following actions:

1. **words = ["AI", "helps", "in", "refactoring", "code"]**: This line creates a Python list named words containing several strings.

2. **sentence = " ".join(words):** This is the key part for string concatenation.

- " " is the separator string. In this case, it's a space.
- .join(words) is a string method that takes an iterable (like our words list) as an argument. It concatenates the elements of the iterable into a single string, using the separator string (" ") between each element.
- The result of this operation is stored in the sentence variable.

3. **print(sentence):** This line prints the final sentence string to the console.

In essence, this code takes the list of words and joins them together with spaces in between to form a single sentence.

TASK – 03:

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

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

OUTPUT :

A terminal window showing the output of the code. On the left, there is a copy icon (two arrows forming a square). To the right of the icon, the text "Not Found" is displayed.

EXPLANATION :

This code snippet does the following:

1. **student\_scores = {"Alice": 85, "Bob": 90}:** This line creates a Python dictionary named student\_scores with two key-

value pairs. The keys are student names (strings), and the values are their scores (integers).

2. **score = student\_scores.get("Charlie", "Not Found"):** This is where the `.get()` method is used.

- `student_scores.get("Charlie", "Not Found")` attempts to retrieve the value associated with the

key "Charlie" from the `student_scores` dictionary.

- The second argument, "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 this operation is stored in the `score` variable.

3. **print(score):** This line prints the value of the `score` variable to the console.

In this specific case, since "Charlie" is not a key in

the `student_scores` dictionary, 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.

TASK – 04:

operation = "multiply" a, b = 5, 3

operations = {

"add": lambda x, y: x + y,

```
"subtract": lambda x, y: x - y, "multiply": lambda x, y: x * y
}
```

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

print(result) OUTPUT :

A terminal window showing the output of the code. It features a light gray border and a dark gray background. On the left side, there is a circular icon with a white double-headed arrow. To the right of the icon, the number "15" is displayed in white text.

EXPLANATION :

This code snippet demonstrates how to use a dictionary to perform different operations based on a given string:

1. **operation = "multiply"**: This line assigns the string "multiply" to the variable operation. This string determines which operation will be performed.
2. **a, b = 5, 3**: This line uses tuple unpacking to assign the values 5 and 3 to the variables a and b respectively. These will be the operands for the chosen operation.
3. **operations = { ... }**: This creates a dictionary named operations.
  - The keys of the 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 defined with

the lambda keyword. Each lambda function takes two arguments (x and y) and performs a specific mathematical operation (addition, subtraction, or multiplication).

4. **result = operations.get(operation, lambda x, y: None)(a, b):** This is the core of the refactored code. Let's break it down:

- **operations.get(operation, lambda x, y: None):** This uses the .get() method on the operations dictionary. It tries to retrieve the value associated with the key stored in the operation variable (which is "multiply" in this case).
  - The second argument, lambda x, y: None, is the default value. If the operation string does not match any key in the operations dictionary, this default lambda function (which simply

returns None) is returned.

- **(a, b):** After the .get() method returns a function (either one of the operation lambda functions or the

default None lambda), this part calls that function with the arguments a and b.

- The result of calling the function is assigned to the result variable.

5. **print(result):** This line prints the final result to the console.

In this specific case,

since operation is "multiply", operations.get("multiply", ...) returns the lambda x, y: x \* y function. This function is then called

with a=5 and b=3, resulting in  $5 * 3 = 15$ , which is stored

in result and then printed. This approach is much more concise

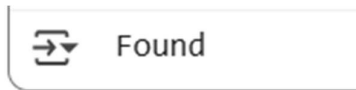
and easier to manage than a long chain of if-elif statements, especially if you have many possible operations.

TASK – 05:

```
items = [10, 20, 30, 40, 50]
```

```
found = 30 in items
```

```
print("Found" if found else "Not Found") OUTPUT :
```



EXPLANATION :

This code snippet checks if a specific value exists within a list and prints a message based on the result:

1. **items = [10, 20, 30, 40, 50]**: This line creates a list named items containing five integer values.
2. **found = 30 in items**: This is the most important part.
  - 30 in items is a Python expression that checks for membership. It evaluates to True if the value 30 is present in the items list, and False otherwise.
  - The result of this check (True or False) is assigned to the boolean variable found.
3. **print("Found" if found else "Not Found")**: This is a Python conditional expression (also known as a ternary operator).
  - It's a concise way to write an if-else statement.
  - It reads as: "Print 'Found' if the found variable is True, otherwise print 'Not Found'."

In this particular case, since 30 is indeed present in the items list, the expression `30 in items` evaluates to `True`. This `True` value is assigned to `found`, and the conditional expression then prints "Found". This is a much more direct and efficient way to check for list membership compared to iterating through the list with a loop.