

AI ASSISTED CODING

LAB ASSIGNMENT-1

Name: Dhommati Akash

HT.NO: 2303A51422

Batch: 21

Question-1: AI-Generated Logic Without Modularization (Factorial without Functions)

Prompt: # generate a code to get factorial of a number without using functions

Code and Output Screenshot:

The screenshot shows a code editor interface with a dark theme. On the left is a sidebar with various icons for file operations like copy, paste, search, and refresh. The main area displays a Python script named 'Task 01.py' with the following code:

```
1 # generate a code to get factorial of a number without using functions
2 number = int(input("Enter a number to find its factorial: "))
3 factorial = 1
4 for i in range(1, number + 1):
5     factorial *= i
6 print(f"The factorial of {number} is {factorial}")
7
```

Below the code editor is a terminal window showing the execution of the script. The terminal output is:

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding & C:\Users\akash\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/akash/OneDrive/Desktop/betech_3_2/Ass coding/Task 01.py"
Enter a number to find its factorial: 6
The factorial of 6 is 720
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding>
```

Observation:

GitHub Copilot was helpful for me being a beginner, it helped me with the right type of logic in loops. It shortened the time to consider syntax and basic control flow logic. Copilot made the things easy like initializing a variable properly and choosing good loop condition expressions. For new user it works more like an intelligent code assistant than an educator. Finally it improves confidence and quickness and must be done while also learning base skills.

Question-2: AI Code Optimization & Cleanup (Improving Efficiency)

Prompt: # generate an optimized version code of Factorial of a given Number.

Code and Output Screenshot:

The screenshot shows a code editor window with two tabs: "Task 01.py" and "Task 01.py > ...". The code in "Task 01.py" is as follows:

```
8
9     # generate an optimized version code of Factorial of a given Number.
10    number = int(input("Enter a number to find its factorial: "))
11    if number < 0:
12        print("Factorial is not defined for negative numbers.")
13    elif number == 0 or number == 1:
14        print(f"The factorial of {number} is 1")
15    else:
16        factorial = 1
17        for i in range(2, number + 1):
18            factorial *= i
19        print(f"The factorial of {number} is {factorial}")
20
21
```

Below the code editor is a terminal window showing the execution of the script and its output:

```
thon.exe "c:/Users/akash/OneDrive/Desktop/betech_3_2/Ass coding/Task 01.py"
Enter a number to find its factorial: 5
The factorial of 5 is 120
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding> & C:\Users\akash\AppData\Local\Programs\ive\Desktop\betech_3_2\Ass coding\Task 01.py"
Enter a number to find its factorial: 7
The factorial of 7 is 5040
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding> []
```

Observation:

Using GitHub Copilot for the optimized factorial code produced a more efficient and well-structured solution. The optimized logic reduced unnecessary computations and improved performance. Copilot suggested clear function design and concise implementation, making the code easy to read and reuse. Inline comments helped explain the optimized approach, encouraging good programming practices.

Question-3: Modular Design Using AI Assistance (Factorial with Functions)

Prompt: # generate a code to get factorial of a number with using functions

Code and Output Screenshot:

The screenshot shows a code editor window with a dark theme. The file is named 'Task 01.py'. The code defines a function 'factorial' that calculates the factorial of a number. It includes comments explaining the logic and handles negative numbers, zero, and one separately. The code then uses a loop to calculate the factorial for positive numbers greater than one. Finally, it prompts the user for input and prints the result. The terminal tab at the bottom shows the execution of the script and its output for the numbers 5 and 6.

```
19
20     # generate a code to get factorial of a number with using functions
21 def factorial(n):
22     if n < 0:
23         return "Factorial is not defined for negative numbers."
24     elif n == 0 or n == 1:
25         return 1
26     else:
27         result = 1
28         for i in range(2, n + 1):
29             result *= i
30     return result
31 number = int(input("Enter a number to find its factorial: "))
32 print(f"The factorial of {number} is {factorial(number)}")
33
34
35
36
37
38
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR

```
The factorial of 5 is 120
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding> & C:\Users\akash\AppData\Local\Programs\Python\Python311\Desktop\betech_3_2\Ass coding\Task 01.py
Enter a number to find its factorial: 6
The factorial of 6 is 720
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding>
```

Observation:

Using GitHub Copilot for a modular design made the code more structured and easier to understand. Copilot suggested meaningful function names and clear parameters, which improves readability. The separation of logic into a function allows the same factorial computation to be reused across multiple programs. Inline comments generated by Copilot helped clarify each step of the logic for beginners. Copilot naturally encourages good programming practices through function-based design.

Question-4: Comparative Analysis – Procedural vs Modular AI Code (With vs Without Functions)

Prompt: No prompt

Code and Output Screenshot: No code

Comparison Table:

| Features | Without Functions | With Functions |
|-------------------|---------------------------|------------------------------|
| Code Structure | Simple and linear | Organized and Modular |
| Length of code | Shorter | Slightly long |
| Reusability | Cannot be reduced easily | Can be reused multiple times |
| Maintenance | Harder for large programs | Easy to debug and modify |
| Calling Mechanism | Runs directly | Function is called |

Technical Report:

or **logic clarity**, a procedural version (without functions) feels simple and direct for very small programs because everything is written in one continuous flow. Beginners can easily follow the steps from input to output. But as the program grows, this style quickly becomes messy and harder to understand. A modular version (using functions) improves clarity by putting the main logic into well-named functions, so anyone reading the code can understand its purpose at a glance.

For **debugging**, procedural code is easy to fix when the program is small. But in longer scripts, finding errors becomes confusing and time-consuming. Modular code makes debugging much easier because problems can usually be traced back to a specific function. This allows developers to test and fix parts of the program independently.

Regarding **AI dependency risk**, both approaches have risks if someone blindly trusts Copilot's suggestions. However, modular code slightly reduces this risk .

Question-5: AI-Generated Iterative vs Recursive Thinking

Iterative:

Prompt: # generate a code to get factorial iteratively

Code and Output Screenshots:

The screenshot shows a terminal window with the following content:

```
Task 01.py X
Task 01.py > ...
33
34
35 # generate a code to get factorial iteratively
36 number = int(input("Enter a number to find its factorial: "))
37 factorial = 1
38 for i in range(1, number + 1):
39     factorial *= i
40 print(f"The factorial of {number} is {factorial}")
41
42
43
```

TERMINAL

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding> & C:\Users\akash\AppData\Local\Programs\ive\Desktop\betech_3_2\Ass coding\Task 01.py"
Enter a number to find its factorial: 4
The factorial of 4 is 24
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding>
```

Recursive:

Prompt: # generate a code to get factorial recursively

Code and Output Screenshots:

```
Task 01.py X
Task 01.py > ...

41
42     # generate a code to get factorial recursively
43     def factorial(n):
44         if n < 0:
45             return "Factorial is not defined for negative numbers."
46         elif n == 0 or n == 1:
47             return 1
48         else:
49             return n * factorial(n - 1)
50 number = int(input("Enter a number to find its factorial: "))
51 print(f"The factorial of {number} is {factorial(number)}")
52
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SQL HISTORY TASK MONITOR

```
hon.exe "c:/Users/akash/OneDrive/Desktop/betech_3_2/Ass coding/Task 01.py"
Enter a number to find its factorial: 5
The factorial of 5 is 120
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding> 7

7
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\Ass coding> █
```

Execution Flow Explanation:

In the **iterative approach**, the program starts with a value of 1 and uses a loop to multiply it with every number from 1 up to the given input. The result is updated step by step inside the same loop until the final factorial value is obtained.

In the **recursive approach**, the function solves the problem by breaking it into smaller parts. Each function call depends on the result of the next call, continuing until it reaches a base case (0 or 1). After reaching the base case, the function calls return one by one, multiplying the values together to produce the final factorial.

Comparative Analysis:

Readability:

The iterative approach is usually easier for beginners to read and understand because the flow of execution is straightforward. Recursive code, although mathematically elegant, can be harder to follow since the function keeps calling itself, which makes tracing the execution more complex.

Stack Usage:

Iterative implementations use constant memory because they rely on a single loop. In contrast, recursive implementations consume extra stack memory for every function call, which increases memory usage.

Performance Implications:

Iterative solutions are generally faster and more memory-efficient. Recursive solutions introduce overhead due to repeated function calls and stack operations, which can slow down execution.

When Recursion Is Not Recommended:

Recursion should be avoided when dealing with very large inputs because it can cause stack overflow. It is also not ideal for performance-critical or memory-limited applications, and when the problem logic does not naturally suit a recursive approach.