

# AIASSISTEDCODING

## LABASSIGNMENT-1

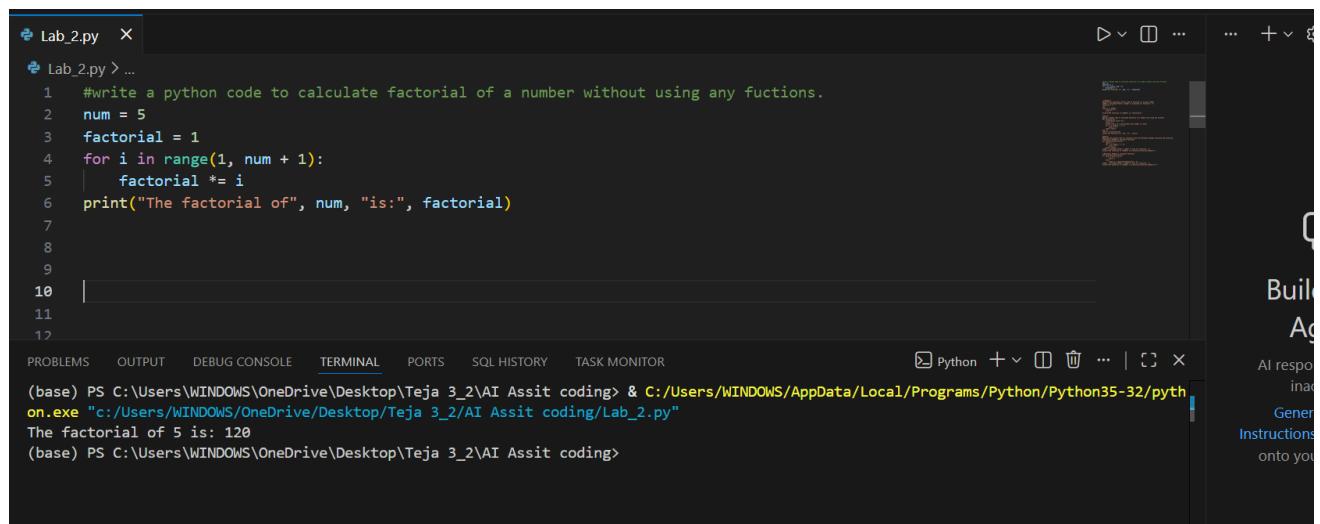
Name:Kashireddy Tejaswini

HT.NO: 2303A51425

Batch:21

**Question-1:**AI-GeneratedLogicWithoutModularization(Factorialwithout Functions)

**Prompt:**#generateacodetogetfactorialofanumberwithoutusingfunctions [Code](#)  
and Output Screenshot:



The screenshot shows a code editor window titled "Lab\_2.py". The code is as follows:

```
1  #write a python code to calculate factorial of a number without using any fuctions.
2  num = 5
3  factorial = 1
4  for i in range(1, num + 1):
5      factorial *= i
6  print("The factorial of", num, "is:", factorial)
7
8
9
10 |
```

Below the code editor, the terminal window shows the output of running the script:

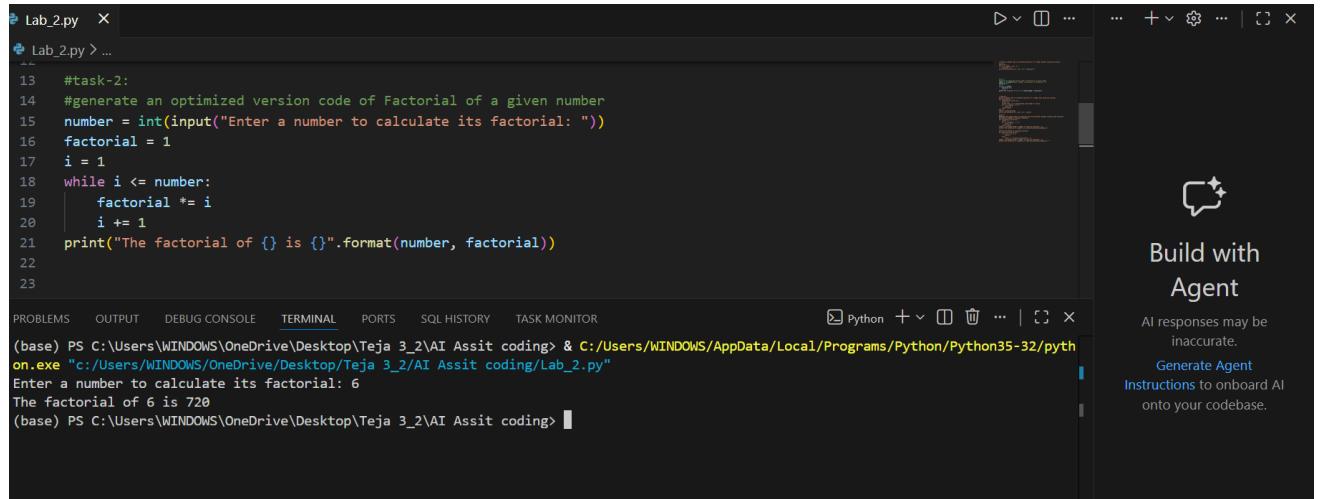
```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/Windows/AppData/Local/Programs/Python/Python35-32/python.exe "c:/Users/Windows/OneDrive/Desktop/Teja 3_2/AI Assit coding/Lab_2.py"
The factorial of 5 is: 120
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit 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 Python code editor in VS Code. The file is named Lab\_2.py. The code defines a function to calculate the factorial of a number using a while loop. The terminal window shows the output of running the script with input 6, which prints "The factorial of 6 is 720". A sidebar on the right is titled "Build with Agent" and includes options for AI responses, generating an agent, and onboard AI instructions.

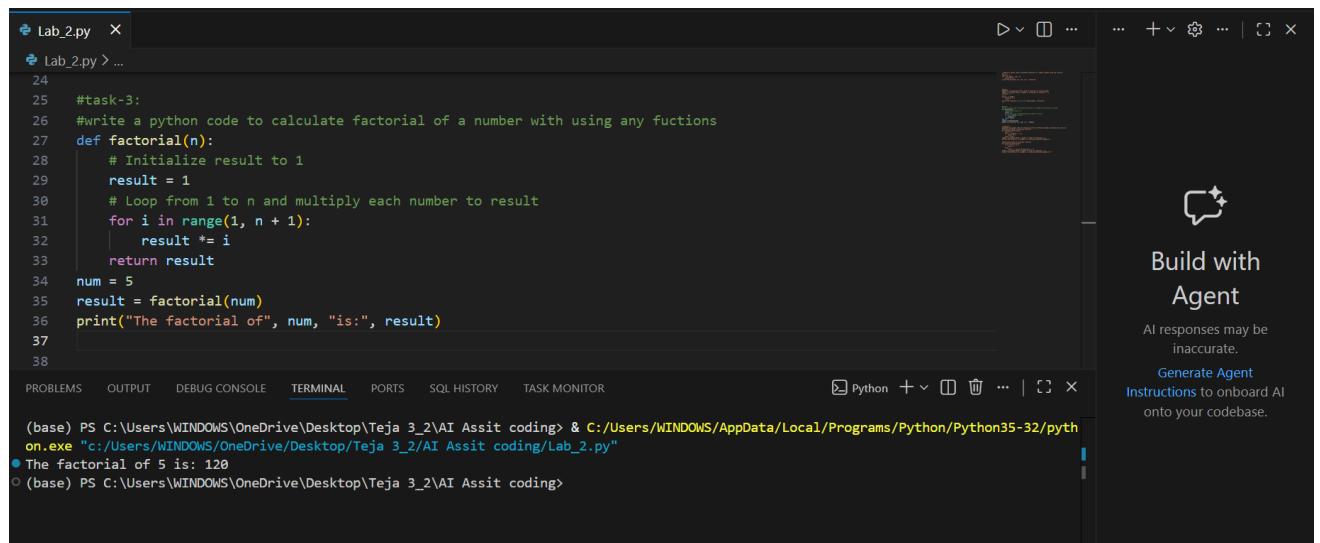
```
13  #task-2:
14  #generate an optimized version code of Factorial of a given number
15  number = int(input("Enter a number to calculate its factorial: "))
16  factorial = 1
17  i = 1
18  while i <= number:
19      factorial *= i
20      i += 1
21  print("The factorial of {} is {}".format(number, factorial))
22
23
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/AppData/Local/Programs/Python/Python35-32/python.exe "c:/Users/WINDOWS/OneDrive/Desktop/Teja 3_2/AI Assit coding/Lab_2.py"
Enter a number to calculate its factorial: 6
The factorial of 6 is 720
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit 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 Python code editor in VS Code. The file is named Lab\_2.py. It contains a function named factorial that calculates the factorial of a number using a for loop. The main part of the code calls this function with num=5 and prints the result. The terminal window shows the output "The factorial of 5 is: 120". A sidebar on the right is titled "Build with Agent" and includes options for AI responses, generating an agent, and onboard AI instructions.

```
24
25  #task-3:
26  #write a python code to calculate factorial of a number with using any fuctions
27  def factorial(n):
28      # Initialize result to 1
29      result = 1
30      # Loop from 1 to n and multiply each number to result
31      for i in range(1, n + 1):
32          result *= i
33      return result
34
35  num = 5
36  result = factorial(num)
37  print("The factorial of", num, "is:", result)
38
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/AppData/Local/Programs/Python/Python35-32/python.exe "c:/Users/WINDOWS/OneDrive/Desktop/Teja 3_2/AI Assit coding/Lab_2.py"
● The factorial of 5 is: 120
○ (base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit 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 (Procedural Code)	With Functions (Modular Code)
Logic Clarity	Logic is written in a single flow, easy to follow for small programs.	Logic is divided into functions, making it clearer and more structured.
Code Structure	Simple and linear structure.	Organized and modular structure.
Reusability	Code cannot be reused easily and may require repetition.	Functions can be reused multiple times in the program.
Debugging Ease	Debugging becomes difficult as the program grows larger.	Easier to debug because errors can be traced to specific functions.
Suitability for Large Projects	Not suitable for large or complex programs.	Well suited for large-scale projects.
AI Dependency Risk	Higher risk due to long blocks of AI-generated code that are harder to understand.	Lower risk because functions encourage understanding and verification of AI-generated logic.

**Technical Report:**

In terms of logic clarity, procedural code written without functions is simple and straightforward for very small programs. Since all instructions are written in a single continuous sequence, beginners can easily follow the execution flow from input to output. This approach helps new programmers understand basic programming concepts without the additional complexity of functions. However, as the program becomes longer and more complex, procedural code starts to lose clarity. Large blocks

of logic reduce readability and make it difficult to understand how different parts of the program are connected. This often leads to confusion and errors during development.

Modular code using functions significantly improves logic clarity by breaking the program into smaller, meaningful units. Each function is designed to perform a specific task, making the code easier to read, understand, and manage. Well-named functions clearly describe their purpose, allowing developers to understand the overall program structure at a glance. This structured approach enhances code organization and makes collaboration easier, especially when multiple developers are working on the same project.

Debugging is another important aspect where modular code performs better than procedural code. Procedural programs are easy to debug only when they are small. As the program grows, locating errors becomes difficult because all logic is tightly coupled in one flow. A single mistake can affect multiple parts of the program, increasing debugging time. In contrast, modular programming simplifies debugging by allowing developers to test and debug individual functions independently. Errors can be quickly traced to a specific function, making troubleshooting faster and more efficient.

Regarding AI dependency risk, both procedural and modular code can be affected if developers blindly trust AI-generated suggestions. However, procedural code poses a higher risk because large blocks of AI-generated logic are harder to verify and understand. Modular code reduces this risk by encouraging developers to review, test, and understand each function separately. This promotes better human oversight and validation of AI-generated code.

Overall, function-based modular programming is more reliable, maintainable, and suitable for large-scale and professional software development.

#### Question-5:AI-GeneratedIterativevsRecursiveThinking

Iterative:

Prompt:#generateacodetogetfactorialiteratively [Code](#)

and Output Screenshots:

A screenshot of the Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, Run, and others. A search bar says "AI Assit coding". The left sidebar shows an "OPEN EDITORS" section with "code.py" and "AI ASSIT CODING" sections. The main editor area contains Python code for calculating a factorial iteratively:

```
1 # Iterative method to calculate factorial using function
2
3 def factorial_iterative(n):
4     result = 1
5     for i in range(1, n + 1):
6         result *= i
7     return result
8
9 num = int(input("Enter a number: "))
10 result = factorial_iterative(num)
11
12 print("The factorial of", num, "is:", result)
13
```

The terminal below shows the output of running the code in a PowerShell window:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/AppData/Local/Programs/Python/Python35-32/python.exe "c:/Users/Windows/OneDrive/Desktop/Teja 3_2/AI Assit coding/code.py"
● Enter a number: 4
The factorial of 4 is: 24
○ (base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

The bottom right corner features an AI assistant interface with a speech bubble icon, "Build with Agent" button, and instructions to onboard AI onto your codebase.

## Recursive:

Prompt:`#generateacodetogetfactorialrecursively` Code

and Output Screenshots:

A screenshot of the Visual Studio Code interface, similar to the previous one but with a different file structure in the Explorer sidebar. The "OPEN EDITORS" section now includes "factorial\_recursive". The main editor area contains Python code for calculating a factorial recursively:

```
1 # Recursive method to calculate factorial using function
2
3 def factorial_recursive(n):
4     if n == 0 or n == 1:
5         return 1
6     return n * factorial_recursive(n - 1)
7
8 num = int(input("Enter a number: "))
9 result = factorial_recursive(num)
10
11 print("The factorial of", num, "is:", result)
12
```

The terminal shows the output of running the code in a PowerShell window:

```
(base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding> & C:/Users/WINDOWS/AppData/Local/Programs/Python/Python35-32/python.exe "c:/Users/Windows/OneDrive/Desktop/Teja 3_2/AI Assit coding/code.py"
● Enter a number: 4
The factorial of 4 is: 24
○ (base) PS C:\Users\WINDOWS\OneDrive\Desktop\Teja 3_2\AI Assit coding>
```

The bottom right corner features an AI assistant interface with a speech bubble icon, "Build with Agent" button, and instructions to onboard AI onto your codebase.

## 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.