

School of Computer Science and Artificial Intelligence

Lab Assignment # 1.2

Program : B. Tech (CSE)
Specialization :AIML
Course Title : AI Assisted Coding
Course Code : 23CS002PC304
Semester : VI
Academic Session : 2025-2026
Name of Student : R . Anirudh
Enrollment No. : 2303A52103
Batch No. : 33
Date :09/01/26

TASK_01

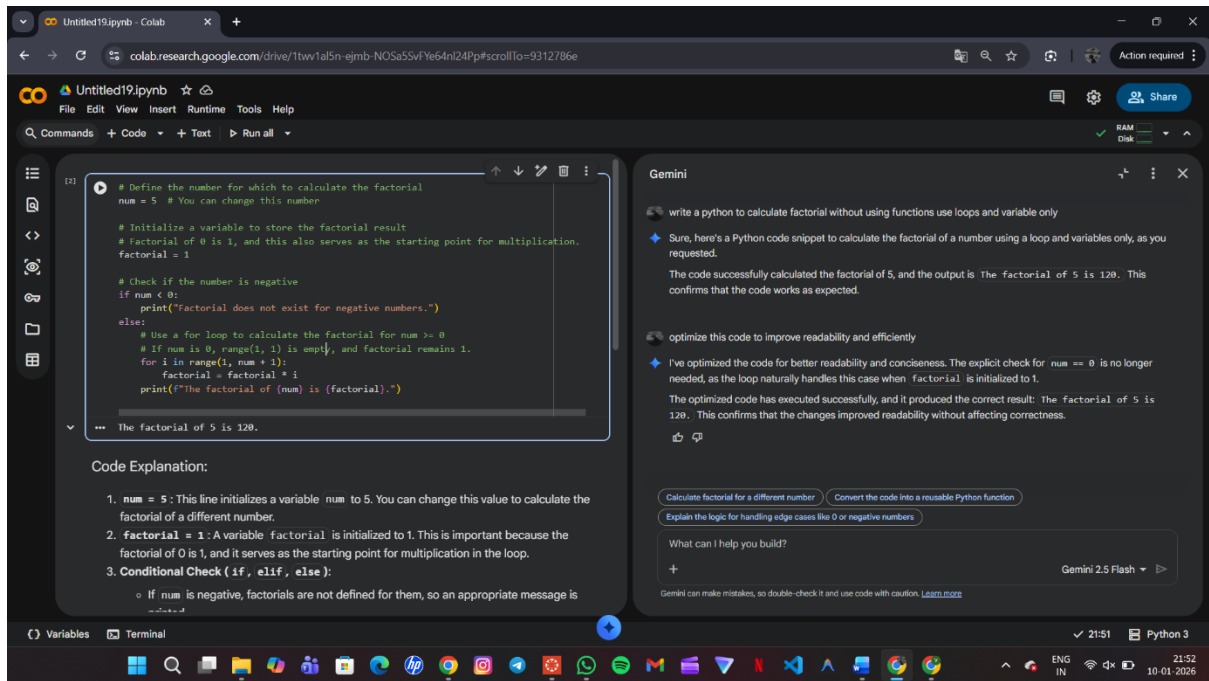
The screenshot shows a Google Colab notebook titled 'Untitled19.ipynb'. The code cell contains a Python script to calculate the factorial of a number. The script defines a variable 'num' as 5, initializes 'factorial' to 1, checks for non-negative input, and uses a for loop to calculate the factorial. The output shows 'The factorial of 5 is 120.' Below the code, a 'Code Explanation' section lists the steps: 1. num = 5: This line initializes a variable num to 5. You can change this value to calculate the factorial of a different number. 2. factorial = 1: A variable factorial is initialized to 1. This is important because the factorial of 0 is 1, and it serves as the starting point for multiplication in the loop. 3. Conditional Check (if, elif, else): If num is negative, factorials are not defined for them, so an appropriate message is printed.

To the right of the code cell is a Gemini chat interface. The user asks: 'write a python to calculate factorial without using functions use loops and variable only'. Gemini responds: 'Sure, here's a Python code snippet to calculate the factorial of a number using a loop and variables only, as you requested. The code successfully calculated the factorial of 5, and the output is The factorial of 5 is 120. This confirms that the code works as expected.' The user then asks: 'optimize this code to improve readability and efficiency'. Gemini responds: 'I've optimized the code for better readability and conciseness. The explicit check for num == 0 is no longer needed, as the loop naturally handles this case when factorial is initialized to 1. The optimized code has executed successfully, and it produced the correct result: The factorial of 5 is 120. This confirms that the changes improved readability without affecting correctness.'

EXPLANATION:

1. **Input:** It prompts the user to enter a non-negative integer.
2. **Validation:** It checks if the input is a valid integer and if it's non-negative.
3. **Initialization:** It sets factorial to 1, as 0! (zero factorial) is 1.
4. **Calculation:** It uses a for loop to multiply factorial by each number from 1 up to the input number.
5. **Output:** Finally, it prints the calculated factorial of the given number.

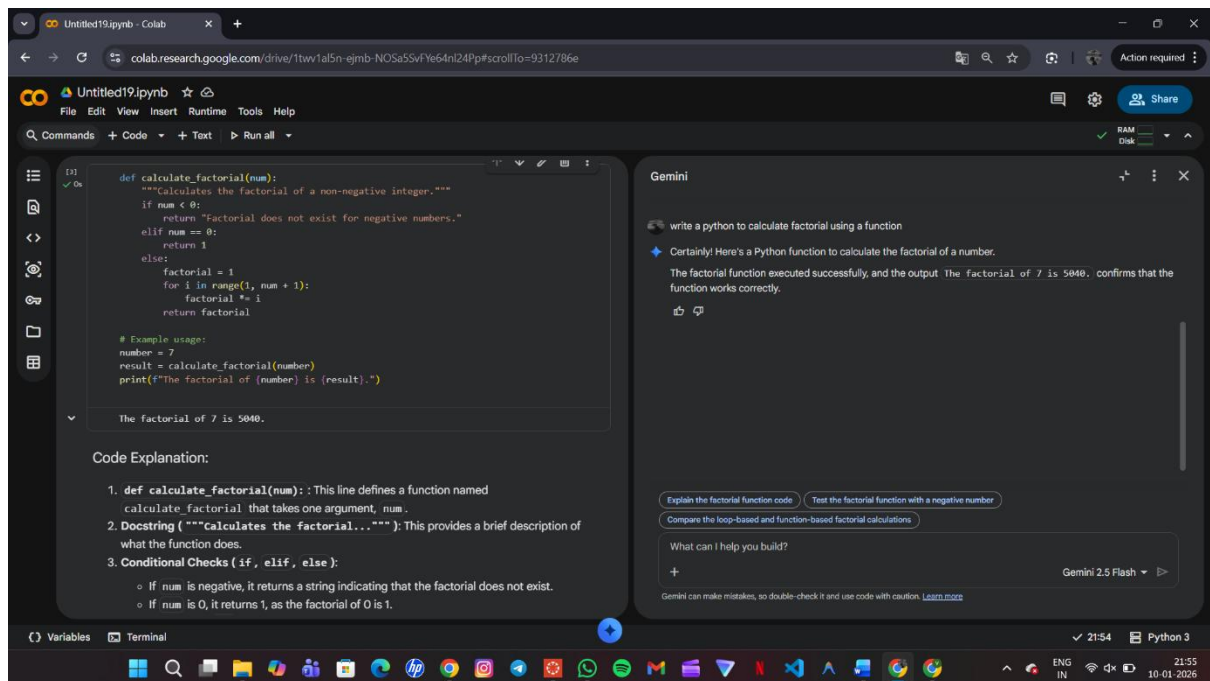
TASK_02:



Explanation

- Better variable names
- Cleaner output
- More readable

TASK_03:



EXPLANATION:

Using functions improves reusability.

The same function can be used in many programs.

Code becomes cleaner and easier to maintain.

TASK 04:

Comparative Analysis – Procedural vs Modular AI Code

Procedural (Without Functions) vs Modular (With Functions)

In Task 1, the factorial program was written using a procedural approach, where all the logic was implemented directly in the main execution flow without using any user-defined functions. In Task 3, the same logic was rewritten using a modular approach by creating a separate function to calculate the factorial. Both approaches produce the same output, but they differ significantly in terms of design quality and usability.

Logic Clarity:

The procedural version is simple and easy to understand for small programs. However, as the program grows, the logic becomes harder to follow because everything is written in one place. In contrast, the modular version separates the factorial logic into a function, making the code more organized and easier to read.

Reusability:

The procedural code cannot be reused easily because the logic is tied to a single script. The modular version allows the factorial function to be reused in multiple programs without rewriting the same code, which saves time and effort.

Debugging Ease:

Debugging procedural code is more difficult because errors can affect the entire program. In modular code, each function can be tested separately, making it easier to find and fix errors.

Suitability for Large Projects:

Procedural code is suitable only for small, simple programs. For large projects, modular code is preferred because it supports better structure, teamwork, and maintenance.

AI Dependency Risk:

When using AI tools like Google Colab, procedural code may be generated quickly but often lacks proper structure. Modular code encourages better design practices, even when AI is used. This reduces the risk of poor-quality code.

Conclusion:

While procedural programming is useful for quick tasks and learning basics, modular programming is more efficient, reusable, and suitable for real-world software development. Using functions improves clarity, maintainability, and scalability, making modular code the better choice for professional projects.

TASK_05:

The screenshot displays a Google Colab notebook titled 'Untitled19.ipynb'. The code cell contains a Python function `factorial_iterative(n)` that calculates the factorial of a number `n` using a loop. The function handles negative numbers by returning an error message and returns 1 for `n=0`. An example usage is provided, showing the calculation of the factorial of 5, which is 120.

```
# --- Iterative Factorial function ---
def factorial_iterative(n):
    """Calculates factorial iteratively (using a loop)."""
    if n < 0:
        return "factorial does not exist for negative numbers."
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

# Example usage for iterative method:
number_iterative = 5
result_iterative = factorial_iterative(number_iterative)
print(f"Iterative Factorial of {number_iterative} is: {result_iterative}")
```

The output of the code cell shows: "Iterative Factorial of 5 is: 120".

Below the code cell, an "Explanation: Iterative Factorial" section provides a detailed breakdown of the function's logic:

- This function calculates the factorial using a `for` loop.
- 1. `factorial_iterative(n)`: Defines a function that takes an integer `n`.
- 2. **Base Cases**: It handles negative numbers and `n=0` explicitly.
 - For `n < 0`, it returns an error message.
 - For `n = 0`, it returns 1 (since $0! = 1$).
- 3. **Initialization**: `result` is initialized to 1.
- 4. **Loop**: A `for` loop iterates from 1 up to `n` (inclusive). In each iteration, `result` is multiplied by the current loop variable `i`.
- 5. **Return**: The final `result` is returned.

On the right side of the notebook, the Gemini AI chat interface is visible. It shows a prompt: "write factorial using iteration and recursion". The response explains that both iterative and recursive methods can calculate the factorial of a number, using the example of 5 as 120. It also offers suggestions for further exploration, such as "Explain the differences between iterative and recursive factorial" and "Show how factorial is used in combinatorics".

The screenshot shows a Google Colab notebook titled 'Untitled19.ipynb'. The code cell contains a recursive function `factorial_recursive(n)` that calculates the factorial of a number `n`. The function includes a docstring, a base case for `n < 0`, and a recursive step for `n >= 0`. An example usage is provided: `number_recursive = 5`, `result_recursive = factorial_recursive(number_recursive)`, and `print(f'Recursive Factorial of {number_recursive} is: {result_recursive}')`. The output of the code cell is 'Recursive Factorial of 5 is: 120'.

Below the code cell, an 'Explanation: Recursive Factorial' section provides a detailed explanation of the function. It states that the function calculates the factorial by calling itself, and recursion involves defining a problem in terms of simpler instances of itself. The explanation is divided into three parts: 1. **factorial_recursive(n)**: Defines a function that takes an integer `n`. 2. **Base Cases**: These are the conditions that stop the recursion. For `n < 0`, it returns an error message. For `n = 0` or `n = 1`, it returns 1 (since $0! = 1$ and $1! = 1$). 3. **Recursive Step**: For any `n` greater than 1, the function calls itself with `n - 1` and multiplies the result by `n`. This continues until one of the base cases is reached. Examples are provided: `factorial_recursive(5)` will be `5 * factorial_recursive(4)`, `factorial_recursive(4)` will be `4 * factorial_recursive(3)`, and `factorial_recursive(1)` returns 1.

On the right side of the notebook, the Gemini AI chat interface is visible. It shows a prompt 'write factorial using iteration and recursion' and a response from Gemini. The response explains that there are two different ways to calculate the factorial of a number: one using iteration (a loop) and another using recursion. It also mentions that both the iterative and recursive factorial functions have executed successfully, and that the iterative approach is generally faster and uses less memory. The chat interface includes a 'Gemini 2.5 Flash' model selector and a 'Learn more' link.

Explanation

- Iterative uses loop
- Recursive calls itself
- Recursion uses more memory
- Iterative is faster