

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 the following Python script:

```
# Define the number for which to calculate the factorial
num = 5 # You can change this number

# Initialize a variable to store the factorial result
factorial = 1

# Check if the number is negative, zero, or positive
if num < 0:
    print("Factorial does not exist for negative numbers.")
elif num == 0:
    print("The factorial of 0 is 1.")
else:
    # Use a for loop to calculate the factorial
    for i in range(1, num + 1):
        factorial *= i
    print(f"The factorial of {num} is {factorial}.")
```

The output of the code is: "... The factorial of 5 is 120."

Code Explanation:

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`)**:
 - o 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 AI interface window. The user asks: "write a python to calculate factorial without using functions use loops and variable only". Gemini responds with a Python code snippet to calculate the factorial of 5 using a loop and variables only. The user then asks: "optimize this code to improve readability and efficiently". Gemini optimizes the code by removing the explicit check for `num == 0` as the loop naturally handles this case when `factorial` is initialized to 1. The optimized code executes successfully and produces 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:

The screenshot shows a Google Colab notebook titled "Untitled19.ipynb". The code cell contains the following Python code:

```
# Define the number for which to calculate the factorial
num = 5 # You can change this number

# Initialize a variable to store the factorial result
# Factorial of 0 is 1, and this also serves as the starting point for multiplication.
factorial = 1

# Check if the number is negative
if num < 0:
    print("Factorial does not exist for negative numbers.")
else:
    # Use a for loop to calculate the factorial. For num >= 0
    # If num is 0, range(1, 1) is empty, and factorial remains 1.
    for i in range(1, num + 1):
        factorial *= i
    print(f"The factorial of {num} is {factorial}.")
```

The output of the code is: "The factorial of 5 is 120."

Below the code, there is a "Code Explanation:" section with the following points:

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):`
o If `num` is negative, factorials are not defined for them, so an appropriate message is printed.

To the right of the code cell, there is a Gemini AI interface. The user asks: "write a python to calculate factorial without using functions use loops and variable only". Gemini responds with the provided code snippet and notes that it successfully calculated the factorial of 5, resulting in 120.

At the bottom of the screen, the taskbar shows various application icons, and the system status bar indicates the date and time as 10-01-2026.

Explanation

- Better variable names
- Cleaner output
- More readable

TASK_03:

The screenshot shows a Google Colab notebook titled "Untitled19.ipynb". The code cell contains a Python function named `calculate_factorial` that calculates the factorial of a non-negative integer. The function includes a docstring, conditional checks for negative numbers and zero, and a loop-based calculation. The output of the code cell shows the factorial of 7 is 5040. Below the code cell is a "Code Explanation:" section with numbered points: 1. Function definition, 2. Docstring, 3. Conditional Checks. The Gemini AI interface is open in a sidebar, asking for a Python function to calculate factorial. It provides a generated function and states it executed successfully with the output 5040 for n=7.

```
def calculate_factorial(num):
    """Calculates the factorial of a non-negative integer."""
    if num < 0:
        return "Factorial does not exist for negative numbers."
    elif num == 0:
        return 1
    else:
        factorial = 1
        for i in range(1, num + 1):
            factorial *= i
        return factorial

# Example usage:
number = 7
result = calculate_factorial(number)
print(f"The factorial of {number} is {result}.")
```

The factorial of 7 is 5040.

Code Explanation:

1. `def calculate_factorial(num):`: This line defines a function named `calculate_factorial` that takes one argument, `num`.
2. `Docstring ("""Calculates the factorial.... """)`: This provides a brief description of what the function does.
3. `Conditional Checks (if, elif, else):`
 - o If `num` is negative, it returns a string indicating that the factorial does not exist.
 - o If `num` is 0, it returns 1, as the factorial of 0 is 1.

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 shows a Google Colab notebook titled "Untitled19.ipynb". The code cell contains an iterative factorial function:

```
# --- 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 is "Iterative Factorial of 5 is: 120". Below the code cell, there is an "Explanation: Iterative Factorial" section with a numbered list:

- This function calculates the factorial using a `for` loop.
- `factorial_iterative(n)`: Defines a function that takes an integer `n`.
- 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 $0! = 1$).
- Initialization**: `result` is initialized to 1.
- Loop**: A `for` loop iterates from 1 up to `n` (inclusive). In each iteration, `result` is multiplied by the current loop variable `i`.
- Return**: The final `result` is returned.

To the right of the Colab interface is a Gemini AI window. The query was "write factorial using iteration and recursion". The response provides both iterative and recursive implementations and compares them:

Certainly! Here are two different ways to calculate the factorial of a number: one using iteration (a loop) and another using recursion.

Both the iterative and recursive factorial functions have executed successfully. You can see that both methods correctly calculated the factorial of 5 as 120. This demonstrates that both iterative and recursive approaches can be used to solve the same problem effectively.

I have provided and explained different ways to calculate the factorial of a number: first, using only loops and variables, then encapsulated in a function, and finally with distinct iterative and recursive function implementations. All the examples have executed successfully. Would you like to explore any other aspects of factorials or perhaps a different programming concept?

The Gemini interface includes a sidebar with buttons for "Explain the differences between iterative and recursive factorial", "Show how factorial is used in combinatorics", and "Suggest a new programming concept to explore". At the bottom, there is a status bar showing "Gemini 2.5 Flash" and system information like "ENG IN", "21:57", and "10-01-2026".

The screenshot shows a Google Colab notebook titled "Untitled19.ipynb". The code cell contains a recursive factorial function:

```
# ... Recursive Factorial Function ...
def factorial_recursive(n):
    """Calculates factorial recursively."""
    if n < 0:
        return "Factorial does not exist for negative numbers."
    elif n == 0 or n == 1:
        return 1
    else:
        # Recursive step: n! = n * (n-1)!
        return n * factorial_recursive(n - 1)

# Example usage for recursive method:
number_recursive = factorial_recursive(5)
print(f"Recursive Factorial of {number_recursive} is: {number_recursive}")

Recursive Factorial of 5 is: 120
```

The output cell shows the result: "Recursive Factorial of 5 is: 120". Below the code cell, there is an explanation of recursion:

Explanation: Recursive Factorial

This function calculates the factorial by calling itself. Recursion involves defining a problem in terms of simpler instances of itself.

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.
 - Example: `factorial_recursive(5)` will be $5 * \text{factorial_recursive}(4)$
 - `factorial_recursive(4)` will be $4 * \text{factorial_recursive}(3)$
 - ... until `factorial_recursive(1)` returns 1.

The right panel of the Colab interface shows a Gemini AI sidebar with the following content:

Gemini

write factorial using iteration and recursion

Certainly! Here are two different ways to calculate the factorial of a number: one using iteration (a loop) and another using recursion.

Both the iterative and recursive factorial functions have executed successfully. You can see that both methods correctly calculated the factorial of 5 as 120. This demonstrates that both iterative and recursive approaches can be used to solve the same problem effectively.

I have provided and explained different ways to calculate the factorial of a number: first, using only loops and variables, then encapsulated in a function, and finally with distinct iterative and recursive function implementations. All the examples have executed successfully. Would you like to explore any other aspects of factorials or perhaps a different programming concept?

What can I help you build?

Explain the differences between iterative and recursive factorial | Show how factorial is used in combinatorics | Suggest a new programming concept to explore

Gemini can make mistakes, so double-check it and use code with caution. Learn more

Explanation

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