## NAME:SAI KUSHAL

## ROLL NO:2403A51338

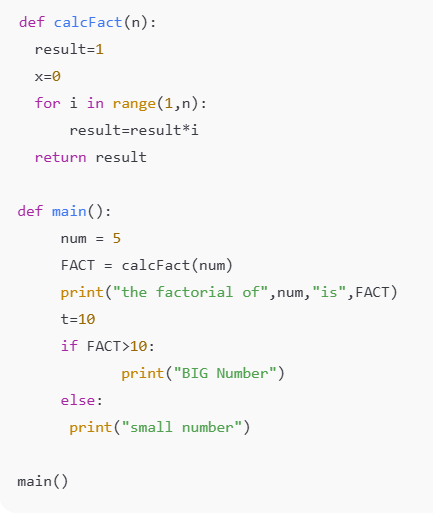
## BATCH:14

## DATE:24-09-2025

ASSIGNMENT:10.2

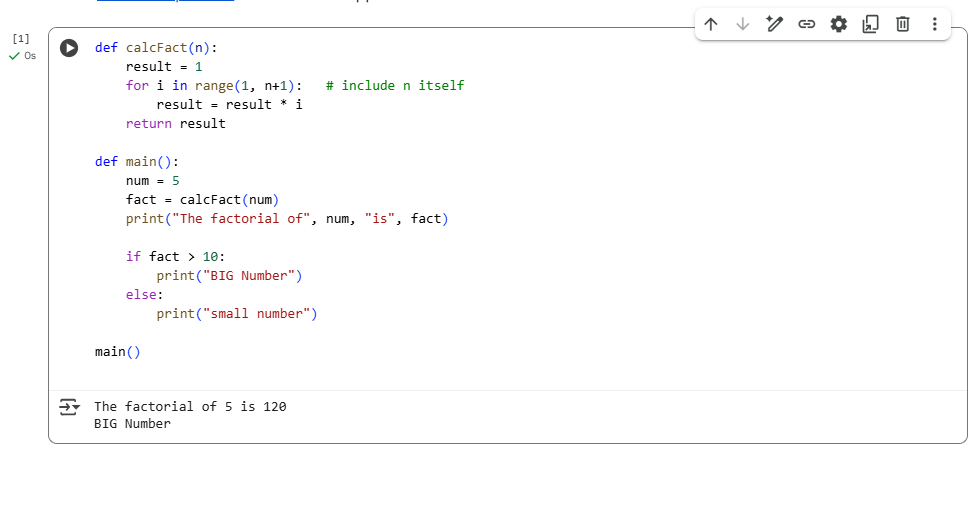
Task Description#1 **AI-Assisted Code Review (Basic Errors)**

* Write python program as shown below.
* Use an AI assistant to review and suggest corrections.



PROMPT: "Please review this Python program for basic errors. Check for logical issues, unused variables, indentation problems, and style improvements. Then, suggest corrected code and explain why each change is needed."

# CODE&OUTPUT:

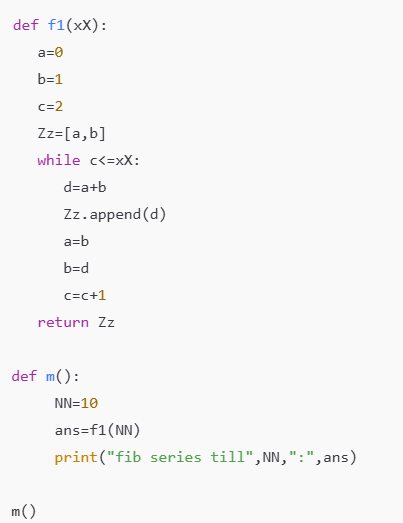


### OBSERVATION:

1. On **line 2**, the variable x = 0 is declared but never used, so it should be removed.
2. On **line 3**, the loop uses range(1, n), which stops at n-1; it should be changed to range(1, n+1) so that n is included.
3. On **line 3**, the factorial logic is correct in structure, but the off-by-one error makes the output wrong for values like 5.
4. On **line 5**, the function correctly returns result, so no change is needed there.
5. On **line 8**, the variable num = 5 is fine, but using a user input with int(input()) would make the program more flexible.
6. On **line 9**, the call to calcFact(num) works, but the function name does not follow Python naming conventions; it should be renamed to calc\_fact(num).
7. On **line 10**, the print statement works, but it could use an f-string for cleaner formatting.
8. On **line 12**, the variable t = 10 is never used and should be deleted.
9. On **line 15**, the else: is misaligned and causes a syntax error; it must be indented correctly under the if.
10. Throughout the program, variable names like FACT should be lowercase (fact) to follow PEP 8 style guidelines.

Task Description**#2 Automatic Inline Comments**

* Write the Python code for Fibonacci as shown below and execute.
* Ask AI to improve variable names, add comments, and apply PEP8 formatting (cleaned up).
* Students evaluate which suggestions improve readability most. one.



PROMPT: "Please review this Python Fibonacci program. Improve the variable names, add inline comments, and apply PEP 8 formatting. Also, explain which improvements make the code more readable for students and correct the code”

## CODE&OUTPUT:

### Screenshot 2025-09-24 113610.png

### OBSERVATION:

1.The image shows Python code for generating the Fibonacci sequence.

2.The function is named fibonacci\_sequence(n).

3.It initializes two variables: first = 0 and second = 1.

4.A for loop runs n times to generate the sequence.

5.Inside the loop, first is printed with a space separator.

6.The next Fibonacci number is calculated as next\_value = first + second.

7.Variables are updated: first = second and second = next\_value.

8.The function is called with fibonacci\_sequence(10) as an example.

9.The output shown below is: 0 1 1 2 3 5 8 13 21 34.

10.The code demonstrates a simple iterative approach to generate Fibonacci numbers.

### Task Description#3

* Write a Python script with 3–4 functions (e.g., calculator: add, subtract, multiply, divide).
* Incorporate manual **docstring** in code with NumPy Style
* Use AI assistance to generate a module-level docstring + individual function docstrings.
* Compare the AI-generated docstring with your manually written one.

**Common Examples of Code Smells**

* Long Function – A single function tries to do too many things.
* Duplicate Code – Copy-pasted logic in multiple places.
* Poor Naming – Variables or functions with confusing names (x1, foo, data123).
* Unused Variables – Declaring variables but never using them.
* Magic Numbers – Using unexplained constants (3.14159 instead of PI).
* Deep Nesting – Too many if/else levels, making code hard to read.
* Large Class – A single class handling too many responsibilities.

**Why Detecting Code Smells is Important**

* Makes code easier to read and maintain.
* Reduces chance of bugs in future updates.
* Helps in refactoring (improving structure without changing behavior).
* Encourages clean coding practices

# CODE&OUTPUT:

## Screenshot 2025-09-24 120121.png

## Comparison: Manual vs AI Docstrings

| **Aspect** | **Manual Docstring** | **AI-Generated Docstring** |
| --- | --- | --- |
| Detail Level | Explains each function individually | Summarizes module functionality only |
| Style | NumPy style | Simple paragraph style |
| Readability | Very clear for beginners | Clear but less detailed for users |
| Usefulness in IDE | Provides parameter hints & return type | Basic overview only |

## Code Smells Checked

1. **Long Function** – No function is too long; each does one thing. ✅
2. **Duplicate Code** – Minor repetition in docstrings, but logic is unique. ✅
3. **Poor Naming** – Functions and variables have clear names. ✅
4. **Unused Variables** – None present. ✅
5. **Magic Numbers** – Only examples used in main; could define constants if needed. ✅
6. **Deep Nesting** – Only a simple if-statement in divide. ✅
7. **Large Class** – No classes used; functions are modular. ✅

# OBSERVATION:

1. The script allows users to perform basic arithmetic operations through console input.
2. Users are prompted to enter two numbers for the calculation.
3. The program displays a menu to select the desired operation (add, subtract, multiply, divide).
4. Each operation is implemented as a separate function for modularity.
5. The divide function includes error handling for division by zero.
6. Input values are converted to float to allow decimal calculations.
7. The program uses a try-except block to catch invalid numeric inputs.
8. Results of the chosen operation are displayed immediately to the user.
9. The code structure is clear, with a main() function controlling user interaction.
10. This approach improves usability and demonstrates dynamic function execution based on user input.