LAB ASSIGNMENT-10.2

< AI Assisted Coding >

NAME : V.KOUSHIK

HALLTICKETNUMBER : 2403A52004

BATCH NUMBER : 01

TASK__1

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

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

Expected Outcome#1: Students need to submit corrected code with comments.

```
def calcFact(n):

result=1

 x=0

for i in range(1,n):

 result=result*i

 return result

def main():

  num = 5

  FACT = calcFact(num)

  print("the factorial of",num,"is",FACT)

  t=10
```

```
    if FACT>10:
        print("BIG Number")
    else:
        print("small number")
main()
```

PROMPT:

Write python program as shown above and submit corrected code with comments.

CODE:

```python
# Define a function to calculate the factorial of a number
def calcFact(n):
    result = 1  # Initialize the result to 1
    # Loop from 1 up to n-1 (inclusive)
    for i in range(1, n + 1): # Corrected the range to include n
        result = result * i  # Multiply the result by the current number in the loop
    return result  # Return the calculated factorial

# Define the main function
def main():
    # Get input from the user and convert it to an integer
    num_str = input("Enter a non-negative integer: ")
    try:
        num = int(num_str)
        if num < 0:
            print("Factorial is not defined for negative numbers.")
            return
    except ValueError:
        print("Invalid input. Please enter an integer.")
        return

    FACT = calcFact(num)  # Call the calcFact function to get the factorial
    print("the factorial of", num, "is", FACT)  # Print the result

    t = 10  # Set a threshold value
    # Check if the factorial is greater than the threshold
    if FACT > t:
        print("BIG Number")  # Print "BIG Number" if factorial is greater than t
    else:
        print("small number")  # Print "small number" otherwise

# Call the main function to run the program
main()
```

```
Enter a non-negative integer: 5
the factorial of 5 is 120
BIG Number
```

Issues

1. **Factorial calculation is incorrect**
   ➤ range(1, n) should be range(1, n + 1) to include n.
2. **Unused variable x = 0**
   ➤ Not used anywhere – remove it.
3. **Unused variable t = 10**
   ➤ Declared but never used – remove it.
4. **Variable name FACT**
   ➤ Uppercase suggests a constant – use lowercase (fact) for readability.
5. **Minor style issues**
   ➤ Print message lacks formatting/capitalization – improve spacing and clarity.

OBSERVATION

The factorial function uses range(1, n), which incorrectly omits the last number; it should be range(1, n + 1).
There are unused variables (x = 0 and t = 10) that should be removed to clean the code.
The variable FACT uses all caps, which is against Python naming conventions for regular variables.
Minor style improvements are needed in print formatting and naming for better readability.

TASK__2

```
def f1(xX):
  a=0
  b=1
  c=2
  Zz=[a,b]
```

```
    while c<=xX:
        d=a+b
        Zz.append(d)
        a=b
        b=d
        c=c+1
    return Zz


def m():
    NN=10
    ans=f1(NN)
    print("fib series till",NN,":",ans)


m()
```

Task Description#2

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

Expected Output#2: Clean format python code with much readability

CODE:

```python
# Function to generate Fibonacci sequence up to 'count' terms
def generate_fibonacci(count):
    first = 0
    second = 1
    index = 2  # Starting from the 3rd term
    fibonacci_series = [first, second]

    # Continue generating terms until the desired count is reached
    while index <= count:
        next_term = first + second
        fibonacci_series.append(next_term)
        first = second
        second = next_term
        index += 1

    return fibonacci_series

# Main function
def main():
    # Get user input and validate it
    try:
        total_terms = int(input("Enter how many Fibonacci terms to generate: "))
        if total_terms < 1:
            print("Please enter a number greater than 0.")
        elif total_terms == 1:
            print("Fibonacci series till 1 term: [0]")
        else:
            result = generate_fibonacci(total_terms - 1)  # Adjust for 0-based indexing
            print(f"Fibonacci series till {total_terms} terms:", result)
    except ValueError:
        print("Invalid input! Please enter a valid integer.")

# Run the program
main()
```

```
Enter how many Fibonacci terms to generate: 6
Fibonacci series till 6 terms: [0, 1, 1, 2, 3, 5]
```

Issues

1. **Poor variable names** – a, b, c, xX, etc., are unclear.
2. **No user input** – Uses a fixed number instead of asking the user.
3. **No input validation** – Doesn't handle invalid or negative values.
4. **Undescriptive function names** – f1() and m() should be meaningful.
5. **No comments or formatting** – Code lacks clarity and PEP8 style.

OBSERVATION:

The code correctly generates the Fibonacci series but uses unclear variable names like a, b, xX, and Zz, which reduce readability.
Functions like f1() and m() are not descriptive and should be renamed.
There is no user input or validation, making the program inflexible and

error-prone.
Additionally, the code lacks comments and does not follow PEP8 formatting guidelines.

TASK__3

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
Dead Code – Code that is never executed.

Expected Output#3: Students learn structured documentation for multi-function scripts

MANUALLY WRITTEN CODE:

```python
"""
simple_calculator.py

A basic calculator module providing fundamental arithmetic operations: add, subtract,
multiply, and divide.

Includes user input handling with validation.
"""

def add(a, b):
    """
    Add two numbers.

    Parameters
    ----------
    a : float
        The first number.
    b : float
        The second number.

    Returns
    -------
    float
        The sum of `a` and `b`.
    """
    return a + b

def subtract(a, b):
    """
    Subtract second number from first.

    Parameters
    ----------
    a : float
        The number from which to subtract.
    b : float
        The number to subtract.

    Returns
    -------
```

```python
    Returns
    -------
    float
        The result of `a - b`.
    """
    return a - b

def multiply(a, b):
    """
    Multiply two numbers.

    Parameters
    ----------
    a : float
        The first number.
    b : float
        The second number.

    Returns
    -------
    float
        The product of `a` and `b`.
    """
    return a * b

def divide(a, b):
    """
    Divide first number by second.

    Parameters
    ----------
    a : float
        The numerator.
    b : float
        The denominator.

    Returns
    -------
    float
        The quotient of `a` divided by `b`.
```

```python
    Raises
    ------
    ValueError
        If `b` is zero.
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

def main():
    """
    Main function to get user input and perform calculations.
    """
    print("Simple Calculator")

    try:
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))
    except ValueError:
        print("Invalid input! Please enter numeric values.")
        return

    print(f"{num1} + {num2} = {add(num1, num2)}")
    print(f"{num1} - {num2} = {subtract(num1, num2)}")
    print(f"{num1} * {num2} = {multiply(num1, num2)}")

    try:
        print(f"{num1} / {num2} = {divide(num1, num2)}")
    except ValueError as e:
        print("Error:", e)

if __name__ == "__main__":
    main()
```

```
Simple Calculator
Enter first number: 4
Enter second number: 5
4.0 + 5.0 = 9.0
4.0 - 5.0 = -1.0
4.0 * 5.0 = 20.0
4.0 / 5.0 = 0.8
```

AI GENERATED CODE:

```python
"""
simple_calculator.py

This module provides basic arithmetic operations: addition, subtraction,
multiplication, and division. It includes user input handling with validation.
"""

def get_float_input(prompt):
    """
    Prompt the user to enter a float and validate the input.

    Parameters:
    prompt (str): The input message displayed to the user.

    Returns:
    float: The validated floating-point number entered by the user.
    """
    while True:
        user_input = input(prompt)
        try:
            return float(user_input)
        except ValueError:
            print("Invalid input. Please enter a numeric value.")

def add(a, b):
    """
    Return the sum of two numbers.

    Parameters:
    a (float): First number.
    b (float): Second number.

    Returns:
    float: Sum of a and b.
    """
    return a + b

def subtract(a, b):
    """
    Return the difference between two numbers.

    Parameters:
    a (float): First number.
    b (float): Second number.
```

```python
    Returns:
    float: Difference of a and b.
    """
    return a - b

def multiply(a, b):
    """
    Return the product of two numbers.

    Parameters:
    a (float): First number.
    b (float): Second number.

    Returns:
    float: Product of a and b.
    """
    return a * b

def divide(a, b):
    """
    Return the division of two numbers.

    Parameters:
    a (float): Numerator.
    b (float): Denominator.

    Returns:
    float: Quotient of a divided by b.

    Raises:
    ValueError: If division by zero is attempted.
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

def main():
    """
    Handle user input and perform arithmetic operations.
    """
    print("Simple Calculator")

    num1 = get_float_input("Enter first number: ")
    num2 = get_float_input("Enter second number: ")
```

```python
def main():
    """
    Handle user input and perform arithmetic operations.
    """
    print("Simple Calculator")

    num1 = get_float_input("Enter first number: ")
    num2 = get_float_input("Enter second number: ")

    print(f"{num1} + {num2} = {add(num1, num2)}")
    print(f"{num1} - {num2} = {subtract(num1, num2)}")
    print(f"{num1} * {num2} = {multiply(num1, num2)}")

    try:
        print(f"{num1} / {num2} = {divide(num1, num2)}")
    except ValueError as e:
        print("Error:", e)

if __name__ == "__main__":
    main()
```

```
Simple Calculator
Enter first number: 7
Enter second number: 8
7.0 + 8.0 = 15.0
7.0 - 8.0 = -1.0
7.0 * 8.0 = 56.0
7.0 / 8.0 = 0.875
```

COMPARISON:

- **Manual docstrings** are detailed, structured, and include sections like Parameters, Returns, and Raises. They are great for clear, professional documentation.
- **AI-generated docstrings** are shorter, simpler, and easier to read quickly but less formal and detailed.
- Manual style is best for large projects, while AI style works well for small scripts or quick docs.

OBSERVATION:

The manual docstrings provide detailed and well-structured documentation, clearly explaining parameters, return values, and exceptions, which is ideal for maintainability and professional projects. In contrast, the AI-generated docstrings are concise and easy to read but lack formal structure and depth. While AI docstrings improve speed and simplicity, manual docstrings offer better clarity for complex codebases.

Choosing between them depends on the project size and documentation needs.