LAB 10.2

Hall Ticket: 2403A51318

Batch: 13

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

Prompt:

Perform an AI-assisted code review for the given Python program. Identify and explain all logical, syntactical, and stylistic errors in paragraph form. Provide the corrected version of the code with clear inline comments describing the modifications. Highlighting the significance of error detection, corrections made, and the importance of following coding standards.

Error Code:

```
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()
```

```
the factorial of 5 is 24
BIG Number
```

Issues Found:

The factorial calculation logic is incorrect because the loop uses range(1, n) which only goes up to n-1, so the last multiplication is skipped. This needs to be changed to range(1, n+1) to include the number n. Additionally, the variable x = 0 is declared but never used in the program, so it is unnecessary and should be removed. Similarly, the variable t = 10 is defined but never used anywhere, making it redundant. Another issue is with naming conventions; the variable FACT should ideally be written in lowercase as fact, and the function name calcFact should be updated to calc_fact to follow Python's PEP8 style guidelines for readability.

Corrected Code:

```python
# Function to calculate factorial of a number
def calc_fact(n):
    result = 1
    # Loop should go till n (inclusive)
    for i in range(1, n + 1):
        result *= i
    return result


def main():
    num = 5
    fact = calc_fact(num)  # Call factorial function
    print("The factorial of", num, "is", fact)

    # Check if factorial is big or small
    if fact > 10:
        print("BIG Number")
    else:
        print("Small Number")


# Call the main function
main()
```

```
The factorial of 5 is 120
BIG Number
```

Observation:

The given task demonstrates the importance of AI-assisted code review in identifying and correcting basic programming errors. The original factorial program contained several issues, such as an incorrect loop range that excluded the last number, unused variables (x and t), and inconsistent naming conventions that reduced readability. Through review, these issues were identified and corrected, resulting in a cleaner and more accurate program. The corrected code now calculates the factorial properly, follows Python standards, and produces the expected output. This exercise highlights how AI support can guide students in debugging, improving code quality, and adhering to coding best practices.

Task Description#2 Automatic Inline Comments

Prompt:

Clean up the given Fibonacci Python code by applying PEP8 formatting, using meaningful variable names, and adding comments or docstrings. Point out the issues in the original code and provide the improved version.

Code:

```
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()
```

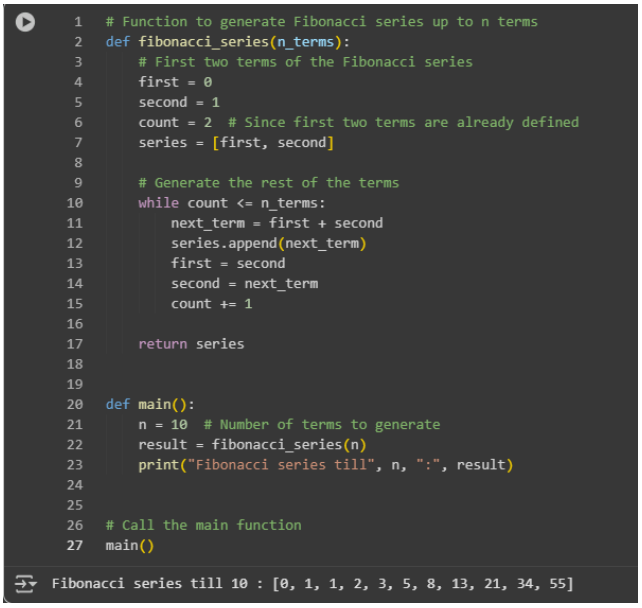
```
fib series till 10 : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Issues Found:

The given Fibonacci program works but has several readability issues. The function and variable names such as f1, xX, Zz, and NN are not meaningful, making it hard to understand the purpose of each element. The code lacks inline comments, which reduces clarity for beginners or anyone reviewing the logic. Additionally, the formatting does not fully follow Python's PEP8 guidelines, as variable names are not in snake_case and spacing is inconsistent. The use of unexplained values such as c = 2 also makes the logic harder to follow without prior knowledge. Overall, while the logic is correct, the program

needs improvements in naming, commenting, and formatting to achieve better readability and maintainability.

Cleaned Code:



```python
# Function to generate Fibonacci series up to n terms
def fibonacci_series(n_terms):
    # First two terms of the Fibonacci series
    first = 0
    second = 1
    count = 2  # Since first two terms are already defined
    series = [first, second]

    # Generate the rest of the terms
    while count <= n_terms:
        next_term = first + second
        series.append(next_term)
        first = second
        second = next_term
        count += 1

    return series


def main():
    n = 10  # Number of terms to generate
    result = fibonacci_series(n)
    print("Fibonacci series till", n, ":", result)


# Call the main function
main()
```

```
Fibonacci series till 10 : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Observation:

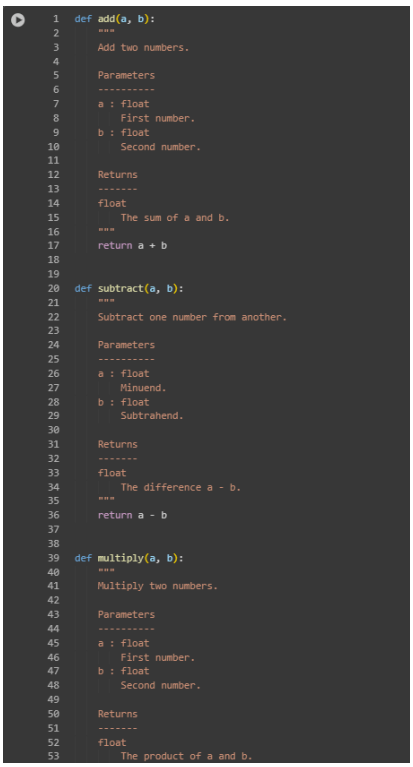
The task of reviewing and improving the Fibonacci program highlights how important code readability and proper formatting are in programming. The original program generated correct results, but its poor variable naming, lack of comments, and inconsistent formatting made it difficult to understand and maintain. After applying meaningful variable names, adding inline comments, and following PEP8 coding standards, the program became much clearer and more professional. This exercise shows that even when logic is correct, clean coding practices are essential for writing programs that are easy to read, debug, and share with others.

Task Description#3

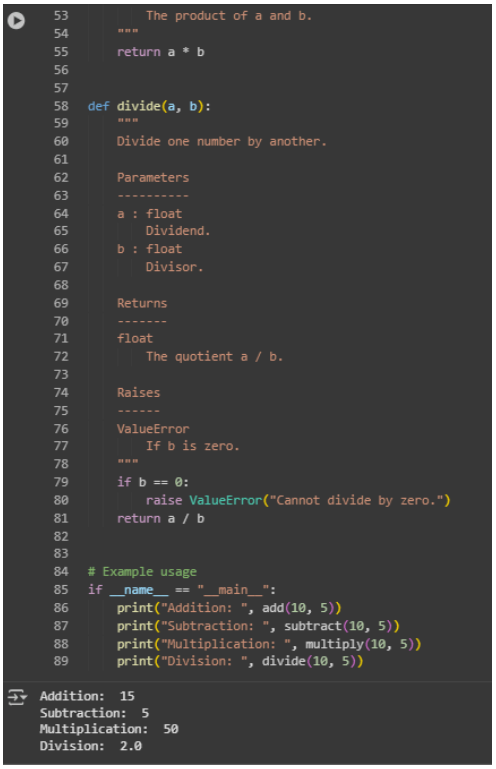
Prompt:

Write a Python program that contains a few calculator functions such as add, subtract, multiply, and divide. For each function, include NumPy-style manual docstrings that explain the parameters, return values, errors, and examples, and also generate AI-style docstrings that are shorter and more natural. Compare the manual docstrings with the AI-generated ones, pointing out the differences in detail and clarity. In addition, explain common examples of code smells such as long functions, duplicate code, poor naming, unused variables, magic numbers, deep nesting, and large classes, and provide simple Python examples to illustrate each of them.

Python Script:



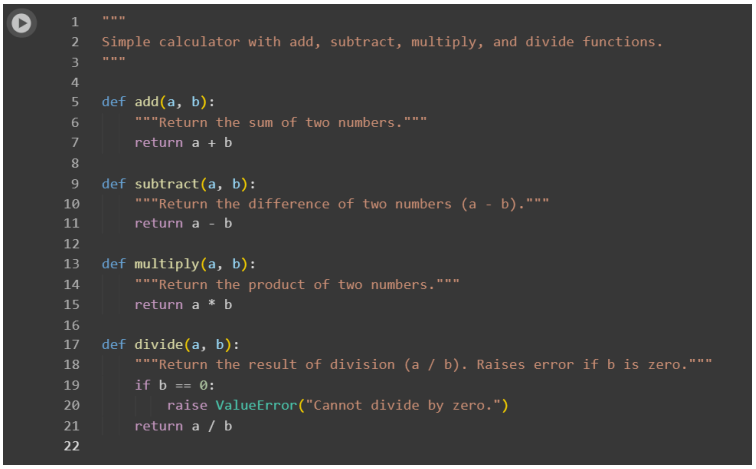
```python
def add(a, b):
    """
    Add two numbers.

    Parameters
    ----------
    a : float
        First number.
    b : float
        Second number.

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


def subtract(a, b):
    """
    Subtract one number from another.

    Parameters
    ----------
    a : float
        Minuend.
    b : float
        Subtrahend.

    Returns
    -------
    float
        The difference a - b.
    """
    return a - b


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

    Parameters
    ----------
    a : float
        First number.
    b : float
        Second number.

    Returns
    -------
    float
        The product of a and b.
```



```python
        The product of a and b.
    """
    return a * b


def divide(a, b):
    """
    Divide one number by another.

    Parameters
    ----------
    a : float
        Dividend.
    b : float
        Divisor.

    Returns
    -------
    float
        The quotient a / b.

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


# Example usage
if __name__ == "__main__":
    print("Addition: ", add(10, 5))
    print("Subtraction: ", subtract(10, 5))
    print("Multiplication: ", multiply(10, 5))
    print("Division: ", divide(10, 5))
```

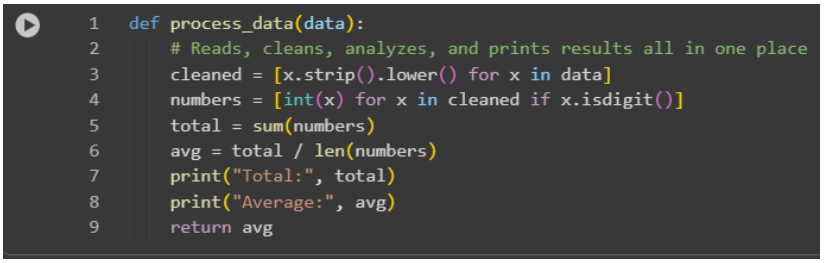
```
Addition:  15
Subtraction:  5
Multiplication:  50
Division:  2.0
```

AI-Generated Version:



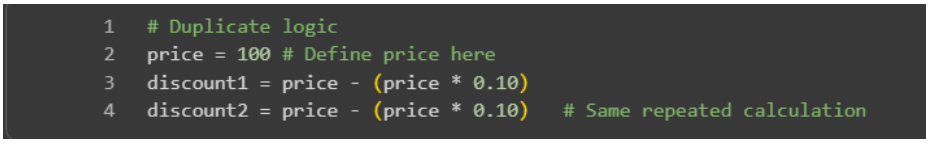
```python
"""
Simple calculator with add, subtract, multiply, and divide functions.
"""

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

def subtract(a, b):
    """Return the difference of two numbers (a - b)."""
    return a - b

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

def divide(a, b):
    """Return the result of division (a / b). Raises error if b is zero."""
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

```

Long Function:



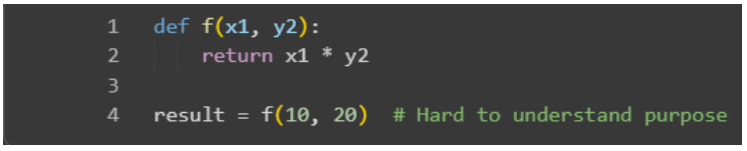
```python
def process_data(data):
    # Reads, cleans, analyzes, and prints results all in one place
    cleaned = [x.strip().lower() for x in data]
    numbers = [int(x) for x in cleaned if x.isdigit()]
    total = sum(numbers)
    avg = total / len(numbers)
    print("Total:", total)
    print("Average:", avg)
    return avg
```

Duplicate Code:



```python
# Duplicate logic
price = 100 # Define price here
discount1 = price - (price * 0.10)
discount2 = price - (price * 0.10)   # Same repeated calculation
```

Poor Naming:



```python
def f(x1, y2):
    return x1 * y2

result = f(10, 20)  # Hard to understand purpose
```

Unused Variables:

```python
x = 100  # Unused variable
y = 50
print(y)
```

```
50
```

Magic Numbers:

```python
PI = 3.14159
r = 5 # Define r here
area = PI * r * r
```

Deep Nesting:

```python
if a > 0:
    if b > 0:
        if c > 0:
            print("All positive")
```

Large Class:

```python
class StudentManager:
    def add_student(self): pass
    def remove_student(self): pass
    def calculate_fees(self): pass
    def print_report(self): pass
    def send_email(self): pass
    # Too many unrelated tasks in one class
```

Comparision:

Comparison: Manual vs AI-Generated Docstrings

Feature	Manual Docstring	AI-Generated Docstring
Module-Level Docstring	Not present	Present: explains module purpose, usage examples
Function Description	More descriptive with context (e.g., "Subtract one number from another")	Short and concise (e.g., "Return the sum of two numbers")
Parameters Section	NumPy-style (`Parameters\n----------`)	Google-style-ish but still close to NumPy format (`Parameters:`)
Returns Section	NumPy-style (`Returns\n-------`)	Inline format (`Returns:`)
Error Handling	Only divide function has `Raises`	Same in divide function
Overall Style	Structured for scientific docs	Clear and concise, more beginner-friendly

Observation:

The AI-generated docstrings provide a clear and concise description of each function and include a module-level docstring that explains the overall purpose and usage of the calculator. Compared to the manually written docstrings, the AI version is shorter, more general, and beginner-friendly, making it easier to quickly understand the functionality. On the other hand, the manual docstrings follow the formal NumPy style, with well-structured sections for parameters, returns, and exceptions, which is more suitable for scientific or professional documentation. Overall, the AI-generated docstrings improve readability and provide context at the module level, while the manual docstrings offer detailed, structured information for precise technical use.