AI ASSIGNMENT: 10

CODE REVIEW AND QUALITY

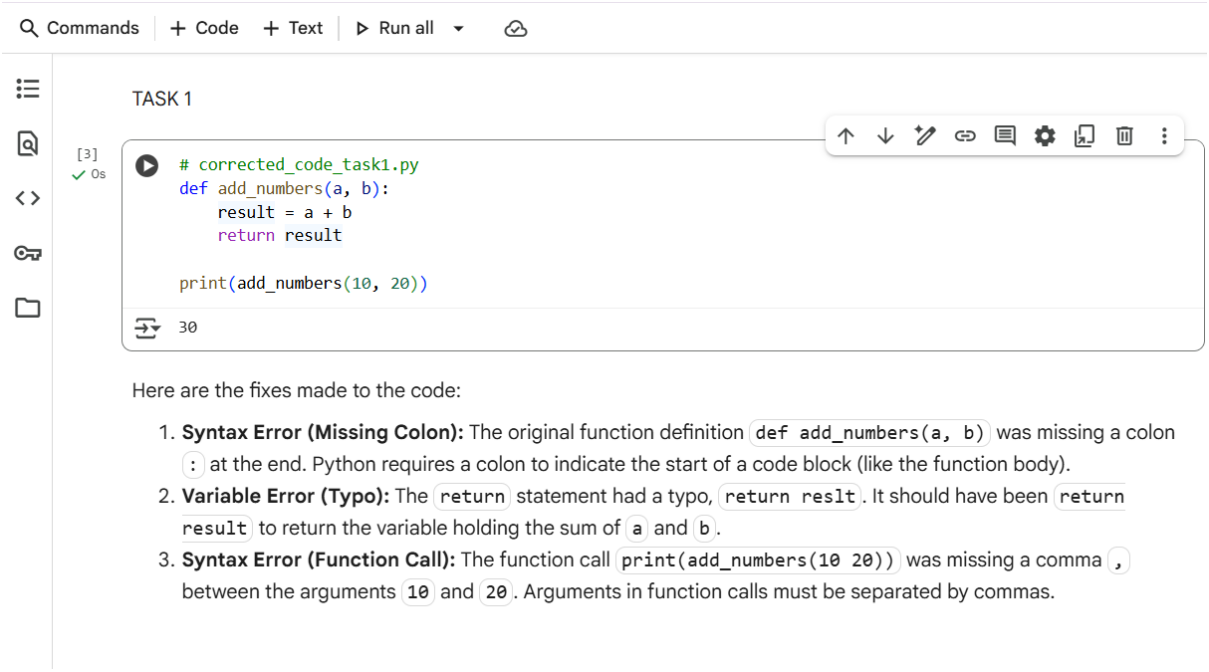
Task 1: Syntax and Error Detection

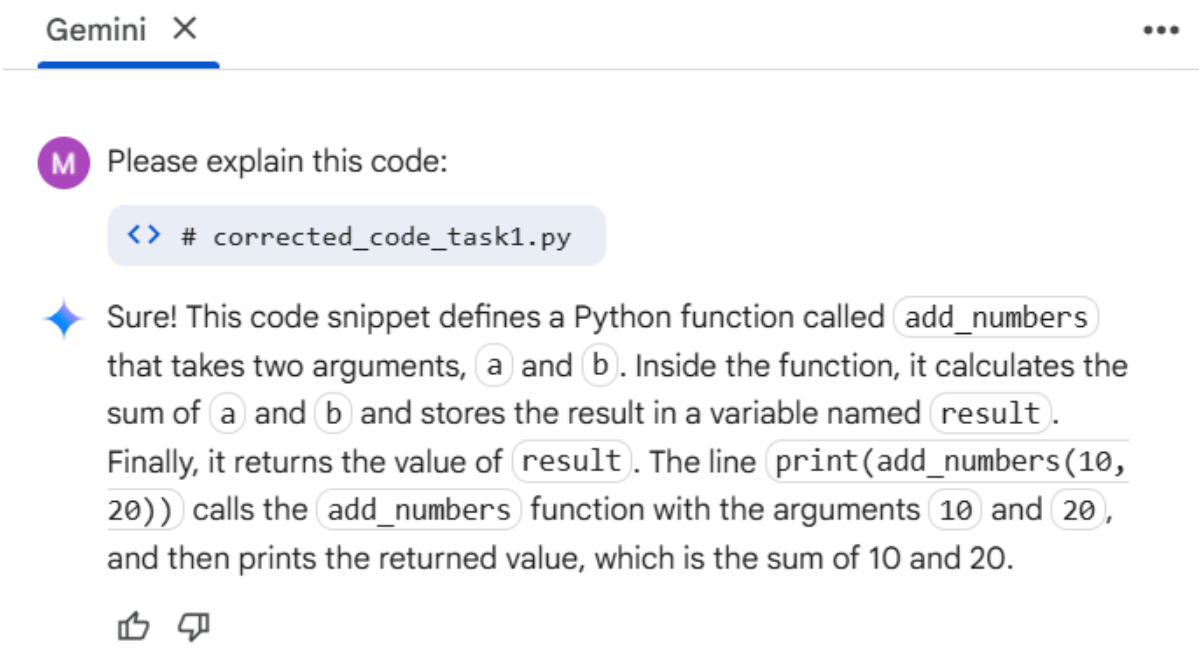
Task: Identify and fix syntax, indentation, and variable errors in the given script.

```
# buggy_code_task1.py
def add_numbers(a, b)
    result = a + b
    return reslt
print(add_numbers(10 20))
```

Expected Output:

- Corrected code with proper syntax (: after function, fixed variable name, corrected function call).
- AI should explain what was fixed.





Task 2: Logical and Performance Issue Review

Task: Optimize inefficient logic while keeping the result correct.

```
# buggy_code_task2.py
def find_duplicates(nums):
    duplicates = []
    for i in range(len(nums)):
        for j in range(len(nums)):
            if i != j and nums[i] == nums[j] and nums[i] not in duplicates:
                duplicates.append(nums[i])
    return duplicates
numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates(numbers))
```

Expected Output:

- More efficient duplicate detection (e.g., using sets).
- AI should explain the optimization.

TASK 2

[5]
✓ 0s

```python
# optimized_code_task2.py
def find_duplicates_optimized(nums):
    seen = set()
    duplicates = set()
    for num in nums:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)
    return list(duplicates)

numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates_optimized(numbers))
```

```
[1, 2]
```

Here is the optimized code and an explanation of the changes:

Explanation of Optimization:

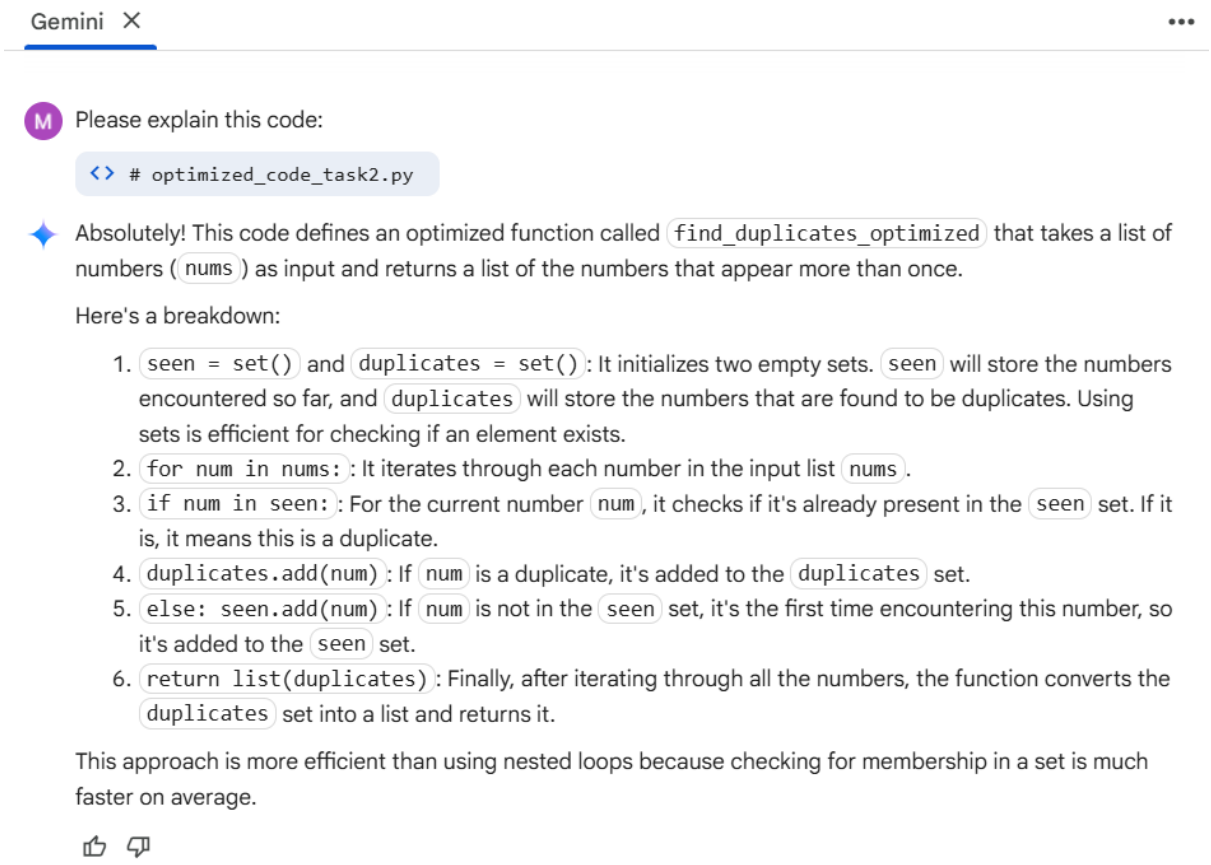
The original code used nested loops, resulting in a time complexity of O(n^2), where n is the number of elements in the list. This becomes very inefficient for large lists.

The optimized code uses sets, which provide O(1) average time complexity for checking membership (`in`) and adding elements (`add`).

Here's how the optimized code works:

1. It initializes two empty sets: `seen` to keep track of the numbers encountered so far, and `duplicates` to store the duplicate numbers.
2. It iterates through the input list `nums`.
3. For each `num` in `nums`:
    - If `num` is already in the `seen` set, it means it's a duplicate, so it's added to the `duplicates` set.
    - If `num` is not in the `seen` set, it's the first time we've seen this number, so it's added to the `seen` set.
4. Finally, it converts the `duplicates` set back to a list and returns it.

This approach has a time complexity of O(n) because each element in the input list is processed a constant number of times. This is significantly more efficient than the original O(n^2) approach.



Task 3: Code Refactoring for Readability

Task: Refactor messy code into clean, PEP 8–compliant, well-structured code.

```
# buggy_code_task3.py

def c(n):
 x=1
 for i in range(1,n+1):
  x=x*i
 return x
print(c(5))
```

Expected Output:

Function renamed to calculate_factorial.

Proper indentation, variable naming, docstrings, and formatting.

AI should provide a more readable version.

TASK 3

```
# refactored_code_task3.py

def calculate_factorial(n):
    """
    Calculates the factorial of a non-negative integer.

    Args:
        n: A non-negative integer.

    Returns:
        The factorial of n.
    """
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result

print(calculate_factorial(5))
```

```
120
```

Gemini

Please explain this code:

```
# refactored_code_task3.py
```

Of course! This code defines a function called `calculate_factorial` that computes the factorial of a non-negative integer `n`.

Here's how it works:

1. **`def calculate_factorial(n):`**: This line defines the function named `calculate_factorial` that accepts one argument, `n`.
2. **Docstring**: The triple-quoted string below the function definition is a docstring. It explains what the function does, its arguments (`Args`), and what it returns (`Returns`). This is a good practice for making code understandable.
3. **`result = 1`**: It initializes a variable called `result` to 1. The factorial of 0 is 1, and this variable will accumulate the product for factorials of positive integers.
4. **`for i in range(1, n + 1):`**: This is a `for` loop that iterates from 1 up to and including `n`.
5. **`result = result * i`**: Inside the loop, in each iteration, the current value of `result` is multiplied by the loop variable `i`, and the product is stored back in `result`. This calculates the running product.
6. **`return result`**: After the loop finishes, the function returns the final value of `result`, which is the factorial of `n`.
7. **`print(calculate_factorial(5))`**: This line calls the `calculate_factorial` function with the argument 5 and prints the returned value (which is the factorial of 5, i.e., 120) to the console.

This refactored code is much more readable than the original version due to the descriptive function name, clear variable names, proper indentation, and the helpful docstring.

Task 4: Security and Error Handling Enhancement

Task: Add security practices and exception handling to the code.

```
# buggy_code_task4.py
import sqlite3
def get_user_data(user_id):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE id = {user_id};"  # Potential
SQL injection risk
    cursor.execute(query)
    result = cursor.fetchall()
    conn.close()
    return result
user_input = input("Enter user ID: ")
print(get_user_data(user_input))
```

Expected Output:

Safe query using parameterized SQL (? placeholders).

Try-except block for database errors.

Input validation before query execution.

TASK 4

[14] ✓ 29s

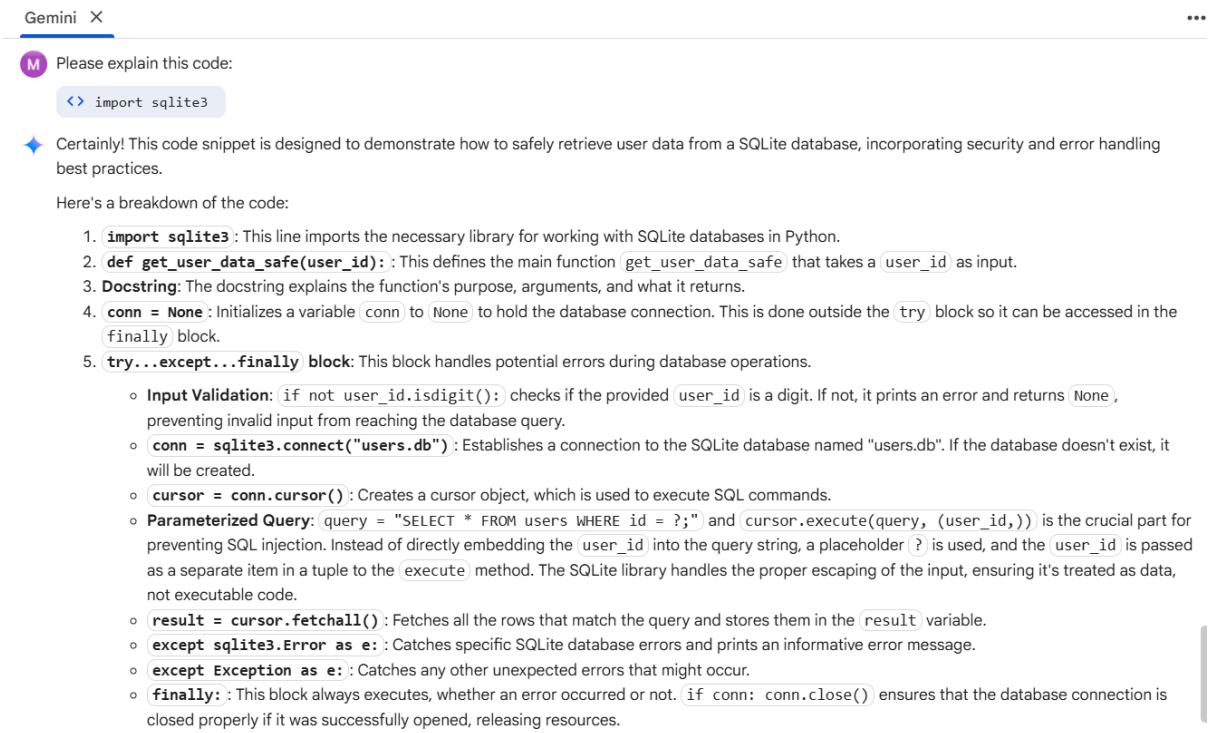
```python
import sqlite3

def get_user_data_safe(user_id):
    """
    Retrieves user data safely from the database.

    Args:
        user_id: The ID of the user to retrieve data for.

    Returns:
        The user data as a list of tuples, or None if an error occurs.
    """
    conn = None  # Initialize connection to None
    try:
        # Input validation: Check if user_id is a digit
        if not user_id.isdigit():
            print("Error: Invalid user ID. Please enter a number.")
            return None

        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()

        # Use parameterized query to prevent SQL injection
        query = "SELECT * FROM users WHERE id = ?;"
        cursor.execute(query, (user_id,)) # Pass user_id as a tuple

        result = cursor.fetchall()
        return result

    except sqlite3.Error as e:
        print(f"Database error: {e}")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None
```

[14] ✓ 29s

```python
    finally:
        if conn: # Close connection if it was successfully opened
            conn.close()

# Example usage:
# First, create a dummy database and table for demonstration if it doesn't exist
try:
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, email TEXT);")
    cursor.execute("INSERT OR IGNORE INTO users (id, name, email) VALUES (101, 'Ishwarya', 'ishwarya@example.com');")
    cursor.execute("INSERT OR IGNORE INTO users (id, name, email) VALUES (201, 'Harsha', 'harsha@example.com');")
    conn.commit()
except sqlite3.Error as e:
    print(f"Error setting up database: {e}")
finally:
    if conn:
        conn.close()


user_input = input("Enter user ID: ")
user_data = get_user_data_safe(user_input)

if user_data:
    if user_data:
        print("User Data:")
        for row in user_data:
            print(row)
    else:
        print("User not found.")
```

```
Enter user ID: 101
User Data:
(101, 'Ishwarya', 'ishwarya@example.com')
```

```
Enter user ID: 201
User Data:
(201, 'Harsha', 'harsha@example.com')
```



Gemini

Please explain this code:

```
import sqlite3
```

Certainly! This code snippet is designed to demonstrate how to safely retrieve user data from a SQLite database, incorporating security and error handling best practices.

Here's a breakdown of the code:

1. `import sqlite3`: This line imports the necessary library for working with SQLite databases in Python.
2. `def get_user_data_safe(user_id):`: This defines the main function `get_user_data_safe` that takes a `user_id` as input.
3. **Docstring**: The docstring explains the function's purpose, arguments, and what it returns.
4. `conn = None`: Initializes a variable `conn` to `None` to hold the database connection. This is done outside the `try` block so it can be accessed in the `finally` block.
5. `try...except...finally` **block**: This block handles potential errors during database operations.
   - **Input Validation**: `if not user_id.isdigit():` checks if the provided `user_id` is a digit. If not, it prints an error and returns `None`, preventing invalid input from reaching the database query.
   - `conn = sqlite3.connect("users.db")`: Establishes a connection to the SQLite database named "users.db". If the database doesn't exist, it will be created.
   - `cursor = conn.cursor()`: Creates a cursor object, which is used to execute SQL commands.
   - **Parameterized Query**: `query = "SELECT * FROM users WHERE id = ?;"` and `cursor.execute(query, (user_id,))` is the crucial part for preventing SQL injection. Instead of directly embedding the `user_id` into the query string, a placeholder `?` is used, and the `user_id` is passed as a separate item in a tuple to the `execute` method. The SQLite library handles the proper escaping of the input, ensuring it's treated as data, not executable code.
   - `result = cursor.fetchall()`: Fetches all the rows that match the query and stores them in the `result` variable.
   - `except sqlite3.Error as e:`: Catches specific SQLite database errors and prints an informative error message.
   - `except Exception as e:`: Catches any other unexpected errors that might occur.
   - `finally:`: This block always executes, whether an error occurred or not. `if conn: conn.close()` ensures that the database connection is closed properly if it was successfully opened, releasing resources.
6. **Example Usage**: The code includes a section to set up a dummy database and table (`users`) with some initial data (including the data for IDs 101 and 201 that were added). This part is enclosed in a `try...except...finally` block as well for robustness.
7. **Getting User Input and Calling the Function**: `user_input = input("Enter user ID: ")` prompts the user to enter an ID. `user_data = get_user_data_safe(user_input)` calls the function with the user's input.
8. **Displaying Results**: The final `if user_data:` blocks check if data was returned and then print the user data or a "User not found" message. The nested `if user_data:` check is slightly redundant but doesn't cause harm.

Task 5: Automated Code Review Report Generation

Task: Generate a **review report** for this messy code.

```
# buggy_code_task5.py

def calc(x,y,z):
 if z=="add":
  return x+y
 elif z=="sub": return x-y
```

```
elif z=="mul":
 return x*y
elif z=="div":
 return x/y
else: print("wrong")

print(calc(10,5,"add"))
print(calc(10,0,"div"))
```

Expected Output:

AI-generated **review report** should mention:

- Missing docstrings
- Inconsistent formatting (indentation, inline return)
- Missing error handling for division by zero
- Non-descriptive function/variable names
- Suggestions for readability and PEP 8 compliance

```python
# corrected_code_task5.py

def perform_arithmetic_operation(num1, num2, operation):
    """
    Performs basic arithmetic operations on two numbers.

    Args:
        num1: The first number.
        num2: The second number.
        operation: A string indicating the operation ("add", "sub", "mul", "div").

    Returns:
        The result of the operation, or an error message for invalid operations or division by zero.
    """
    if operation == "add":
        return num1 + num2
    elif operation == "sub":
        return num1 - num2
    elif operation == "mul":
        return num1 * num2
    elif operation == "div":
        if num2 == 0:
            return "Error: Division by zero is not allowed."
        else:
            return num1 / num2
    else:
        return "Error: Invalid operation."

print(perform_arithmetic_operation(10, 5, "add"))
print(perform_arithmetic_operation(10, 0, "div"))
print(perform_arithmetic_operation(10, 5, "mul"))
print(perform_arithmetic_operation(10, 5, "sub"))
print(perform_arithmetic_operation(10, 5, "mod"))
```

```
15
Error: Division by zero is not allowed.
50
5
Error: Invalid operation.
```

TASK 5

Code Review Report for `buggy_code_task5.py`

Based on the provided code snippet, here is a review highlighting areas for improvement:

Issues Identified:

- **Missing Docstrings:** The function `calc` lacks a docstring. Docstrings are essential for explaining what the function does, its parameters, and what it returns.
- **Inconsistent Formatting:**
  - **Indentation:** The indentation is inconsistent, particularly within the `if`, `elif`, and `else` blocks. PEP 8 recommends consistent 4-space indentation.
  - **Inline Return:** The `elif z=="sub": return x-y` uses an inline return statement, which can make the code less readable compared to having the return statement on a new line.
- **Missing Error Handling for Division by Zero:** The code does not handle the case where the operation is "div" and the divisor `y` is zero. This will result in a `ZeroDivisionError`.
- **Non-descriptive Function/Variable Names:**
  - The function name `calc` is very short and doesn't clearly indicate what kind of calculation it performs (it handles multiple arithmetic operations). A more descriptive name like `perform_arithmetic_operation` would be better.
  - The variable names `x`, `y`, and `z` are not descriptive. They should be renamed to reflect their purpose (e.g., `num1`, `num2`, `operation`).

Suggestions for Readability and PEP 8 Compliance:

- **Add a Docstring:** Include a clear and concise docstring at the beginning of the `calc` function explaining its purpose, arguments, and return value.
- **Fix Indentation:** Ensure consistent 4-space indentation throughout the function.
- **Use Separate Lines for Return Statements:** Place each `return` statement on its own line for improved readability.
- **Implement Error Handling for Division by Zero:** Add a check within the `if z=="div":` block to see if `y` is zero. If it is, handle this case gracefully, perhaps by returning an error message or raising a specific exception.
- **Use Descriptive Names:** Rename the function and variables to be more descriptive and reflect their roles in the code.
- **Adhere to PEP 8:** Review and apply other PEP 8 guidelines for code style and formatting to ensure consistency and readability.

By addressing these points, the code will become more understandable, maintainable, and less prone to errors.

Gemini ×

Please explain this code:

`# corrected_code_task5.py`

Certainly! This code defines a Python function called `perform_arithmetic_operation` which is a refactored and improved version of the `calc` function from Task 5.

Here's a breakdown:

1. `def perform_arithmetic_operation(num1, num2, operation):` : This line defines the function, taking three arguments: `num1` (the first number), `num2` (the second number), and `operation` (a string indicating the desired arithmetic operation). The name `perform_arithmetic_operation` is more descriptive than the original `calc`.
2. **Docstring**: The docstring clearly explains what the function does, its arguments, and what it returns, including the error messages.
3. `if operation == "add": return num1 + num2` : This checks if the `operation` is "add". If it is, it returns the sum of `num1` and `num2`.
4. `elif operation == "sub": return num1 - num2` : If the operation is not "add", it checks if it's "sub" and returns the difference.
5. `elif operation == "mul": return num1 * num2` : If the operation is not "add" or "sub", it checks if it's "mul" and returns the product.
6. `elif operation == "div":` : If the operation is "div", it enters this block to handle division.
    - `if num2 == 0:` : Inside the "div" block, it checks if the second number (`num2`) is zero.
    - `return "Error: Division by zero is not allowed."` : If `num2` is zero, it returns an informative error message instead of causing a `ZeroDivisionError`.
    - `else: return num1 / num2` : If `num2` is not zero, it performs the division and returns the result.
7. `else: return "Error: Invalid operation."` : If the `operation` string does not match any of the supported operations ("add", "sub", "mul", "div"), it returns an "Invalid operation" error message.
8. `print(...)` **lines**: These lines demonstrate how to call the `perform_arithmetic_operation` function with different numbers and operations, including cases that should return a valid result, the division by