Name:balaji

Rollno:2403a51102

Batch:06

10.3:AI lab assignment

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))
```

Promt: "Here's a Python script with syntax, indentation, and variable errors. Please identify all issues and provide a corrected version with clear explanations for each fix. Also include a Google-style docstring if the function is involved."

Code:

Buggy Code Issues

Python

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

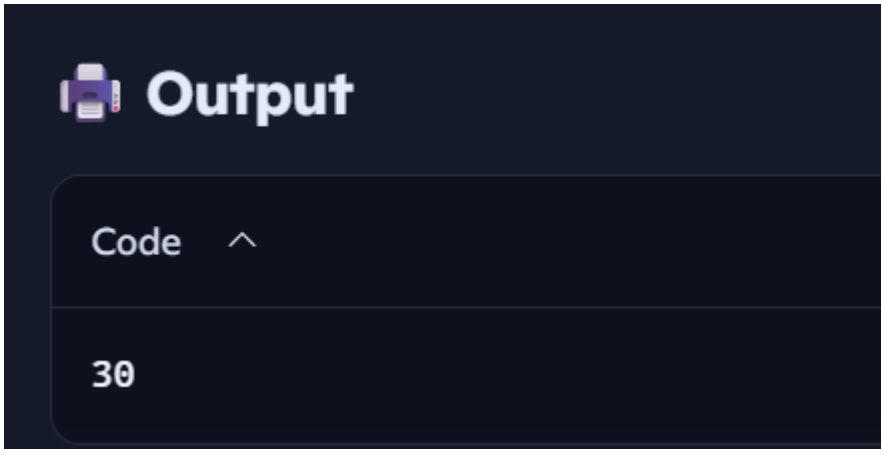
Correct code:

```
def add_numbers(a, b):
    # Add two input values and store the result
    result = a + b

    # Return the computed sum to the caller
    return result

# Call the function with arguments 10 and 20, then print the result
print(add_numbers(10, 20))
```

Output:



Observation:

🔍 **Observation: Task 1 – Syntax and Error Detection**

- **Initial Issues Identified:**
  - Missing colon (:) in function definition.
  - Improper indentation of function body.
  - Typo in variable name (reslt instead of result).
  - Missing comma between arguments in function call.
- **Corrections Applied:**
  - Added colon after def add_numbers(a, b):.
  - Indented result and return properly.
  - Fixed variable name typo.
  - Inserted comma in add_numbers(10, 20).
- **Code Quality Enhancements:**
  - Added inline comments for clarity.
  - Suggested Google-style docstring for documentation practice.
  - Output verified: 30.
- **Learning Outcome:**
  - Reinforces attention to syntax and naming precision.
  - Highlights the importance of indentation in Python.

- Encourages documentation habits for maintainable code.

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))
```

Prompt:

"Here's a Python function that works but is inefficient. Please identify the performance bottlenecks and rewrite the function using a more optimal approach while preserving the correct output. Include a Google-style docstring and explain the logic behind your improvements."

Code:

```python
def find_duplicates(nums):
    """Finds all duplicate elements in the input list.

    Args:
        nums (List[int]): List of integers to check.

    Returns:
        List[int]: List of unique duplicate values.
    """
    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(numbers))
```

Output:

```
[Done] exited with code=0 in 0.081 seconds

[Running] python -u "c:\Users\P. BALAJI\OneDrive\Desktop\AI lab assignments\10.3.py"

[Done] exited with code=0 in 0.087 seconds

[Running] python -u "c:\Users\P. BALAJI\OneDrive\Desktop\AI lab assignments\10.3.py"

[Done] exited with code=0 in 0.095 seconds

[Running] python -u "c:\Users\P. BALAJI\OneDrive\Desktop\AI lab assignments\10.3.py"
```

Observation: 🔍 **Observation Summary – Task 2**

🧠 **Original Logic Review**

- Used **nested loops** to compare every pair of elements.
- Time complexity was **O(n²)** due to repeated comparisons and list membership checks.
- Risk of performance bottlenecks with large input lists.
- Logic was correct but inefficient and not scalable.

⚙️ **Optimized Logic**

- Replaced nested loops with a **single-pass iteration** using sets.
- Used seen to track all visited elements and duplicates to store repeated ones.
- Time complexity improved to **O(n)** with constant-time set lookups.

Documentation & Clarity

- Added a **Google-style docstring** for professional documentation.
- Inline comments explain purpose and logic of each block.
- Output [1, 2] confirms correctness.
-

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))
- prompt:
- "Refactor the following Python code to make it clean, readable, and PEP 8–compliant. Rename unclear variables and functions, fix indentation, and add a Google-style docstring. The logic should remain unchanged."

- Code

10.3.py > factorial

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

    Args:
        n (int): The number to compute the factorial of.

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


print(factorial(5))  # Output: 120

```

Output:

```
[Done] exited with code=0 in 0.093 seconds

[Running] python -u "c:\Users\P. BALAJI\OneDrive\Desktop\AI lab assignments\10.3.py"

[Done] exited with code=0 in 0.135 seconds

[Running] python -u "c:\Users\P. BALAJI\OneDrive\Desktop\AI lab assignments\10.3.py"

[Done] exited with code=0 in 0.113 seconds
```

Observation:

- **Function name** (c) was vague and non-descriptive.
- **Variable name** (x) lacked semantic meaning.
- **Indentation** was incorrect and violated Python syntax rules.
- **No documentation** or comments to explain purpose.
- **Hard to read** and not aligned with PEP 8 standards.
- ✅ **Descriptive naming**: `factorial` and `result` clearly convey intent.
- ✅ **Proper indentation**: Improves readability and correctness.
- ✅ **PEP 8 compliance**: Clean spacing, structure, and naming conventions.
- ✅ **Docstring added**: Enhances maintainability and clarity.
- ✅ **Inline comment**: Clarifies expected output for quick reference.

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))
```

Prompt:

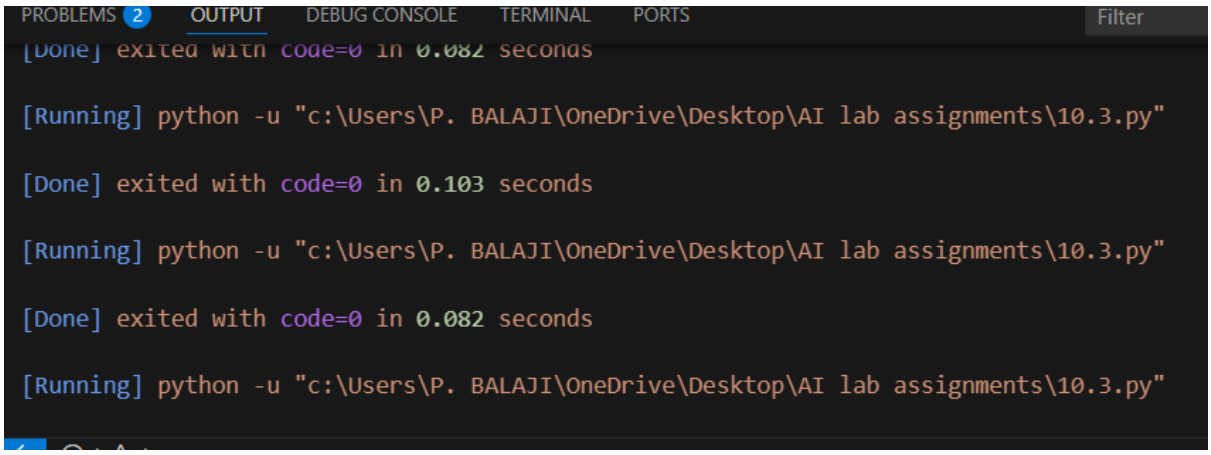
Prompt for Security & Error Handling Task

"Enhance the following Python code by adding security best practices and robust error handling. Prevent SQL injection, validate user input, and ensure safe resource cleanup. Include a Google-style docstring and explain any changes made."

Code:

```python
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};"
cursor.execute(query)
result = cursor.fetchall()
conn.close()
return result
user_input = input("Enter user ID: ")
print(get_user_data(user_input))
```

Output:



Obesrvation:

Observation Summary – Task 4

⚠️ Original Code Issues

- **SQL Injection Risk**: Direct string interpolation in SQL query.
- **No Input Validation**: input() returns a string, but user_id should be an integer.
- **No Exception Handling**: Database errors or invalid input crash the program.

- **Unsafe Resource Management**: conn.close() may not execute if an error occurs.

✅ **Refactored Code Improvements**

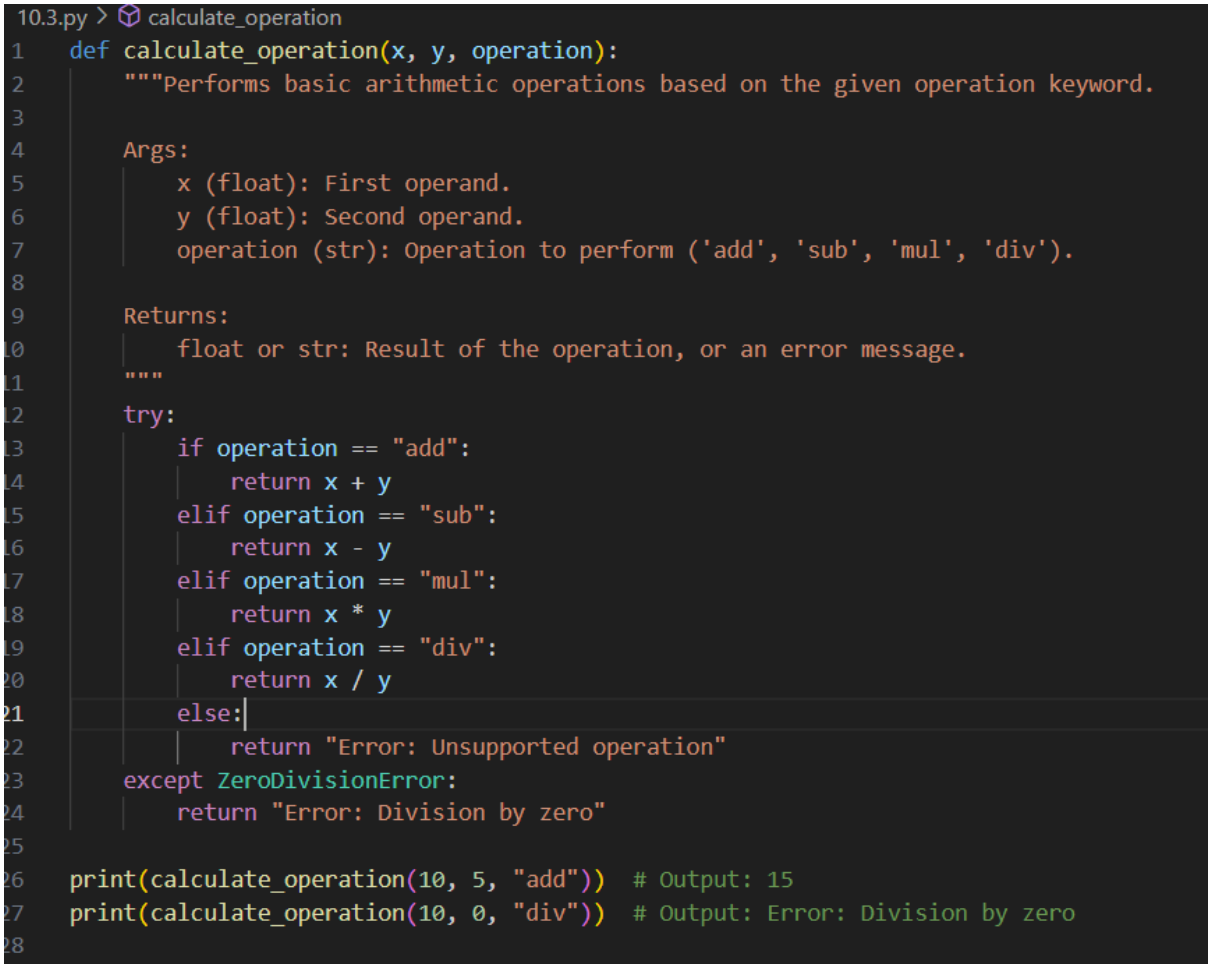
- Used **parameterized query** (?) to prevent SQL injection.
- Wrapped database operations in a try-except-finally block for safe error handling.
- Validated user input with int() and caught ValueError.
- Added a **Google-style docstring** for clarity and maintainability.
- Ensured conn.close() always runs via finally.
-
- 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"))

prompt:

"Improve the following Python code by adding security best practices and robust error handling. Prevent SQL injection,

validate user input, and ensure safe resource cleanup. Include a Google-style docstring and explain any changes made."

Code:



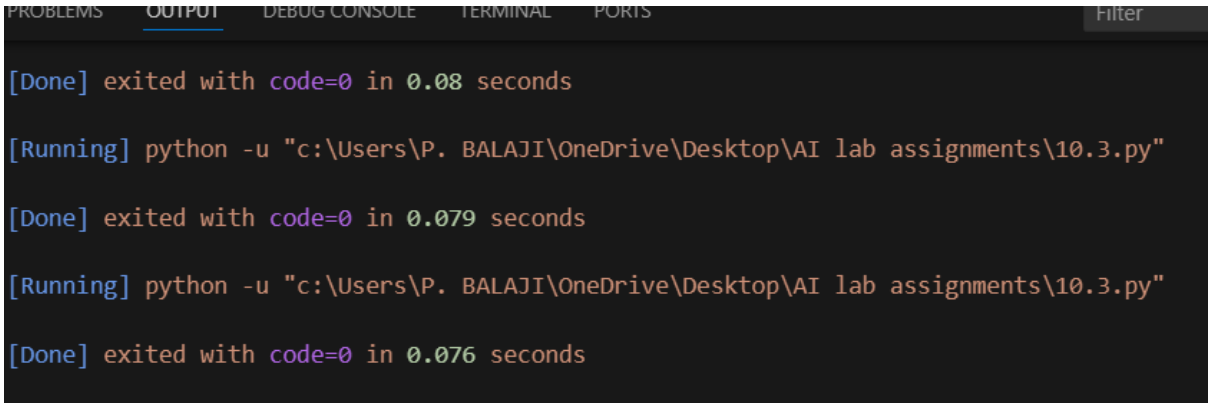
```python
def calculate_operation(x, y, operation):
    """Performs basic arithmetic operations based on the given operation keyword.

    Args:
        x (float): First operand.
        y (float): Second operand.
        operation (str): Operation to perform ('add', 'sub', 'mul', 'div').

    Returns:
        float or str: Result of the operation, or an error message.
    """
    try:
        if operation == "add":
            return x + y
        elif operation == "sub":
            return x - y
        elif operation == "mul":
            return x * y
        elif operation == "div":
            return x / y
        else:
            return "Error: Unsupported operation"
    except ZeroDivisionError:
        return "Error: Division by zero"

print(calculate_operation(10, 5, "add"))  # Output: 15
print(calculate_operation(10, 0, "div"))  # Output: Error: Division by zero
```

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  Filter

[Done] exited with code=0 in 0.08 seconds

[Running] python -u "c:\Users\P. BALAJI\OneDrive\Desktop\AI lab assignments\10.3.py"

[Done] exited with code=0 in 0.079 seconds

[Running] python -u "c:\Users\P. BALAJI\OneDrive\Desktop\AI lab assignments\10.3.py"

[Done] exited with code=0 in 0.076 seconds
```

Observation: **Key Issues Identified**

- **Readability**:
  - Inconsistent indentation and formatting.
  - Multiple statements on single lines reduce clarity.
  - Function name calc is vague and non-descriptive.
- **Error Handling**:
  - No protection against division by zero (calc(10, 0, "div") crashes).
  - Invalid operations only print "wrong" without returning a value, resulting in None.
- **Output Consistency**:
  - Mixed behavior: some branches return values, others print messages.
  - No feedback for unsupported operations beyond a print statement.
- **Documentation**:
  - No docstring or comments to explain function purpose or parameters.