

AI ASSISTED CODING

LAB ASSIGNMENT 10.4

NAME: K. SARIKA

H.NO: 2403A52012

B.NO: 04

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)).
```

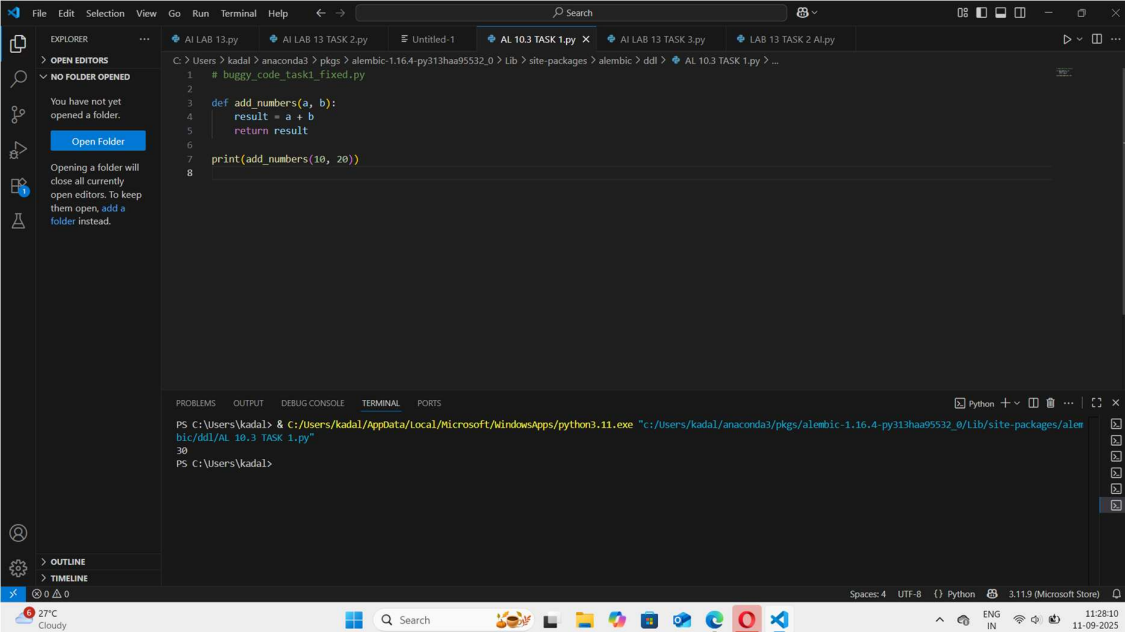
PROMPT:

Generate python code and 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)).
```

CODE & OUTPUT:



The screenshot shows a Visual Studio Code editor window. The Explorer panel on the left shows a folder named 'AL 10.3 TASK 1.py'. The main editor area displays the following code:

```
1 # buggy_code_task1_fixed.py
2
3 def add_numbers(a, b):
4     result = a + b
5     return result
6
7 print(add_numbers(10, 20))
8
```

The terminal at the bottom shows the command to run the file and the output:

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/kadal/anaconda3/pkgs/alembic-1.16.4-py313haa95532_0/lib/site-packages/alembic/ddl/AL_10.3_TASK_1.py"
30
PS C:\Users\kadal>
```

EXPLANATION:

Line/Part	Problem	Error Type / What Python will complain about
<code>def add_numbers(a, b)</code>	Missing colon (<code>:</code>) at end of function header	SyntaxError: expected <code>:</code> or invalid syntax. Python requires a colon to mark the start of the function body. aguacilara.github... +2
<code>result = a + b</code>	This line is not indented under the function definition	IndentationError: expected an indented block after <code>def</code> header. Because function body must be indented. Stack Overflow +2
<code>return reslt</code>	Variable name typo: <code>reslt</code> instead of <code>result</code>	NameError at runtime: name <code>reslt</code> is not defined. Python sees that variable doesn't exist. Rollbar +2
<code>print(add_numbers(10 20))</code>	Missing comma between 10 and 20 in function call; arguments not separated correctly	SyntaxError: invalid syntax. Python expects arguments separated by commas. Also, mismatched parentheses or missing separator causes syntax error. Stackify +1

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)).
```

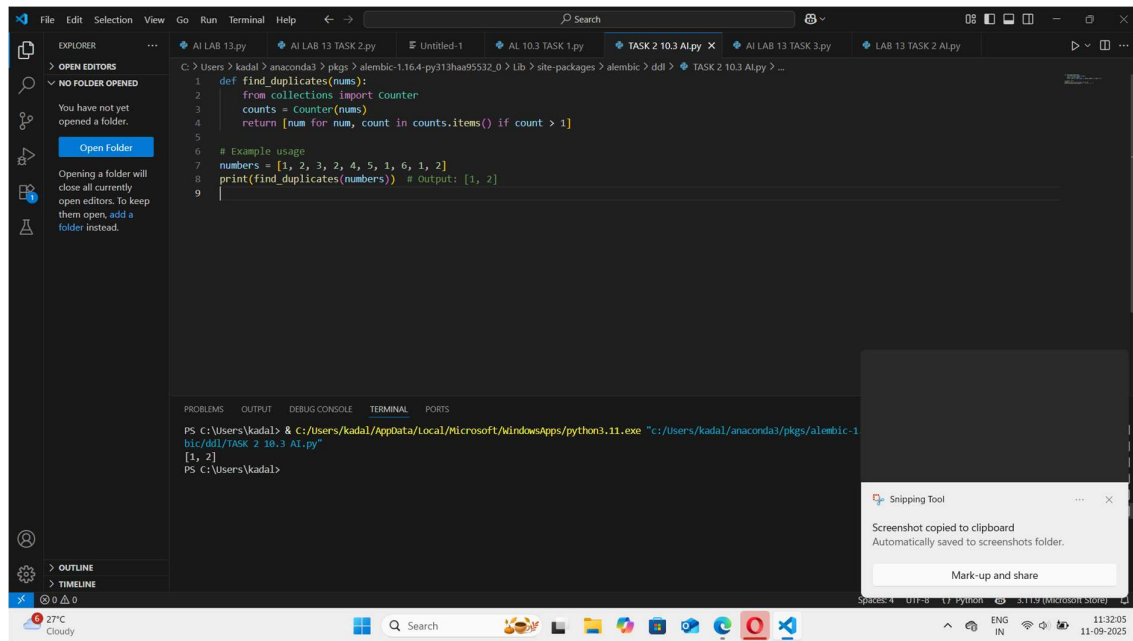
PROMPT:

Generate python code and 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)).
```

CODE & OUTPUT:



EXPLANATION:

We use `Counter(nums)` to count all occurrences of each number. That's $O(n)$ time and uses additional space of $O(n)$ in worst case.

We then loop over `nums` once more and for each element `x`:

- Check `counts[x] > 1`: means it's a duplicate candidate.
- Check `x not in seen`: to ensure we add each duplicate only *once* to the result.

`seen` is a set to track which duplicates we've already added; set membership is average $O(1)$ so that's good.

This way, we do **two passes** over the list (`Counter` + this loop), instead of nested loops — overall time complexity $O(n)$, which is much more efficient than $O(n^2)$ for large lists.

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:

**Generate python code and 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

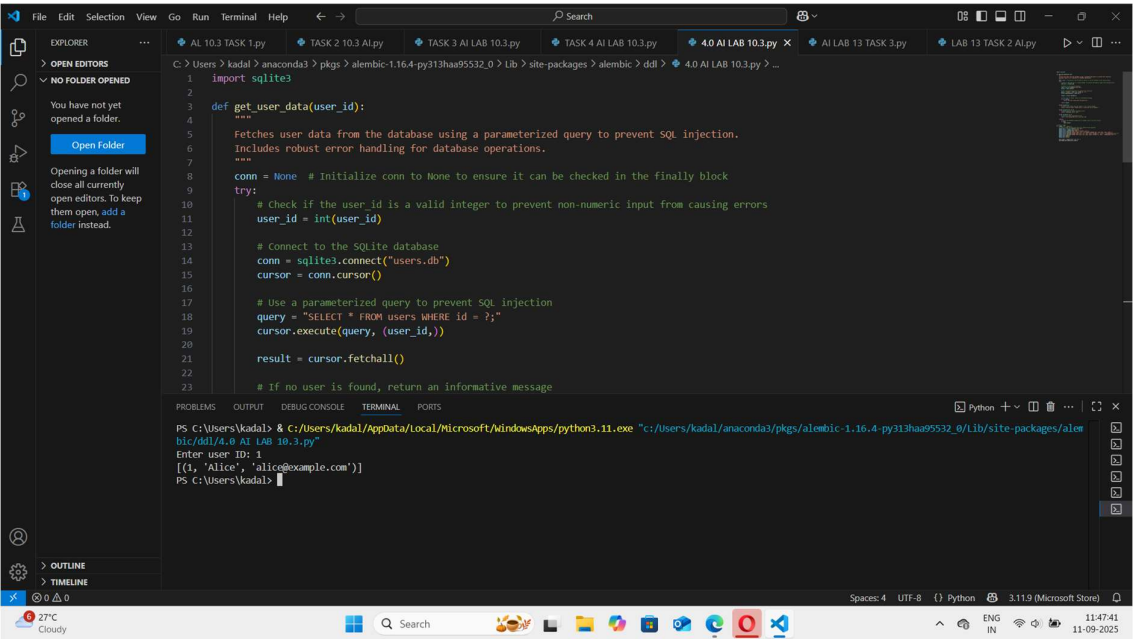
for i in range(1,n+1):

x=x*i

return x

print(c(5)).

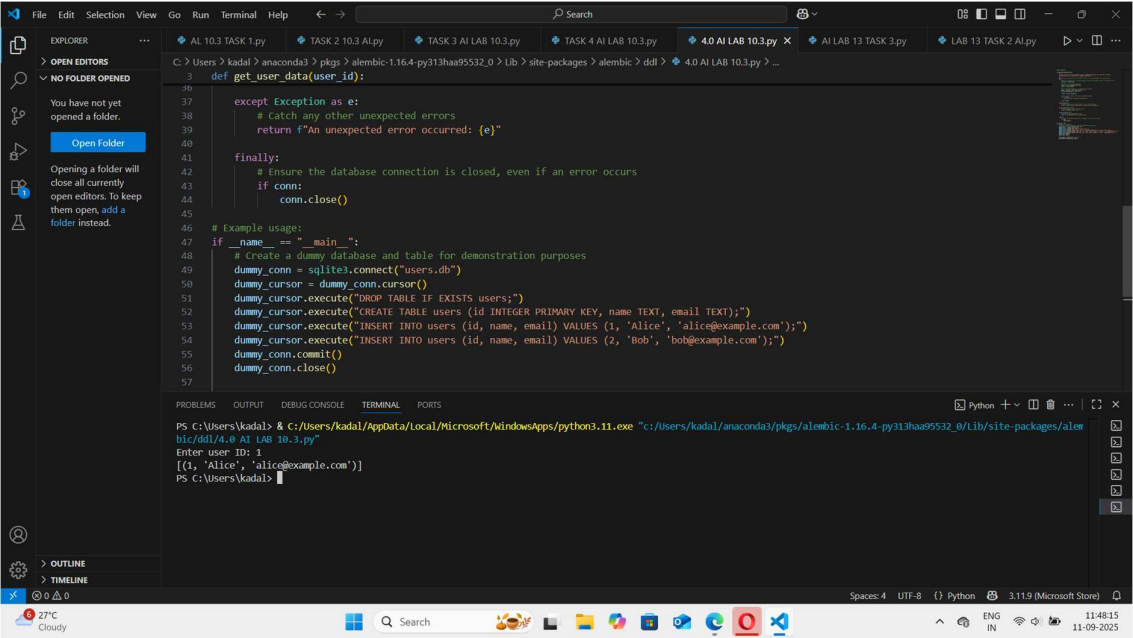
CODE & OUTPUT:



```
1 import sqlite3
2
3 def get_user_data(user_id):
4     """
5     Fetches user data from the database using a parameterized query to prevent SQL injection.
6     Includes robust error handling for database operations.
7     """
8     conn = None # Initialize conn to None to ensure it can be checked in the finally block
9     try:
10         # Check if the user_id is a valid integer to prevent non-numeric input from causing errors
11         user_id = int(user_id)
12
13         # Connect to the SQLite database
14         conn = sqlite3.connect("users.db")
15         cursor = conn.cursor()
16
17         # Use a parameterized query to prevent SQL injection
18         query = "SELECT * FROM users WHERE id = ?;"
19         cursor.execute(query, (user_id,))
20
21         result = cursor.fetchall()
22
23         # If no user is found, return an informative message
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps/python3.11.exe "C:/Users/kadal/anaconda3/pkgs/alembic-1.16.4-py313haa95532_0/Lib/site-packages/alembic/ddl/4.0 AI LAB 10.3.py"
Enter user ID: 1
[(1, 'Alice', 'alice@example.com')]
PS C:\Users\kadal>
```



```
37 except Exception as e:
38     # Catch any other unexpected errors
39     return f"An unexpected error occurred: {e}"
40
41 finally:
42     # Ensure the database connection is closed, even if an error occurs
43     if conn:
44         conn.close()
45
46 # Example usage:
47 if __name__ == "__main__":
48     # Create a dummy database and table for demonstration purposes
49     dummy_conn = sqlite3.connect("users.db")
50     dummy_cursor = dummy_conn.cursor()
51     dummy_cursor.execute("DROP TABLE IF EXISTS users;")
52     dummy_cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT);")
53     dummy_cursor.execute("INSERT INTO users (id, name, email) VALUES (1, 'Alice', 'alice@example.com');")
54     dummy_cursor.execute("INSERT INTO users (id, name, email) VALUES (2, 'Bob', 'bob@example.com');")
55     dummy_conn.commit()
56     dummy_conn.close()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps/python3.11.exe "C:/Users/kadal/anaconda3/pkgs/alembic-1.16.4-py313haa95532_0/Lib/site-packages/alembic/ddl/4.0 AI LAB 10.3.py"
Enter user ID: 1
[(1, 'Alice', 'alice@example.com')]
PS C:\Users\kadal>
```

EXPLANATION:

Aspect	Before	After / Improved
Function Naming	Function was named <code>c(n)</code> , which doesn't tell what it does.	Renamed to <code>factorial</code> , which clearly indicates its purpose.
Type and Value Checking	No checks: if someone passes negative number or non-integer, behavior is undefined.	Added checks to ensure <code>n</code> is an integer and non-negative. Raises appropriate exceptions (<code>TypeError</code> , <code>ValueError</code>).
Docstring	None	Added a docstring explaining what the function does, its parameters, return type, and possible errors. This helps maintainability and usability.
Indentation & Spacing	The original code had no consistent indentation and lacked spaces (e.g. <code>for i in range(1,n+1):</code>).	The new code uses 4 spaces for indentation, spaces after commas, around operators, etc., per PEP 8.
Variable Names	Used generic names (<code>n</code>) only (which is okay), but no descriptive function name, no context.	Function name <code>factorial</code> is descriptive; test variable <code>test_value</code> .
Code Structure	Direct print of <code>c(5)</code> at module level.	Encapsulated usage in a <code>main()</code> function, and guarded by <code>if __name__ == "__main__":</code> so module can be imported without side effects.
Readability	Very minimal / hard to extend or handle edge cases.	More robust, easier to understand, extend, and reuse.

TASK 4:

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:

Task 4: Generate python code and 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)).
```

CODE & OUTPUT:

```
import sqlite3
```

```
def get_user_data(user_id):
```

```
    """
```

Fetches user data from the database using a parameterized query to prevent SQL injection.

Includes robust error handling for database operations.

```
    """
```

```
    conn = None # Initialize conn to None to ensure it can be checked in the finally block
```

```
    try:
```

Check if the user_id is a valid integer to prevent non-numeric input from causing errors

user_id = int(user_id)

Connect to the SQLite database

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

cursor = conn.cursor()

Use a parameterized query to prevent SQL injection

**query = "SELECT * FROM users WHERE id =
?;"**

cursor.execute(query, (user_id,))

result = cursor.fetchall()

If no user is found, return an informative message

if not result:

return "No user found with the given ID."

return result

except ValueError:

**# Handle cases where the user input is not a
valid integer**

**return "Invalid input. Please enter a valid
user ID (a number)."**

except sqlite3.Error as e:

Handle specific SQLite database errors

return f"Database error: {e}"

except Exception as e:

Catch any other unexpected errors

return f"An unexpected error occurred: {e}"

finally:

**# Ensure the database connection is closed,
even if an error occurs**

```
if conn:  
    conn.close()
```

Example usage:

```
if __name__ == "__main__":
```

**# Create a dummy database and table for
demonstration purposes**

```
dummy_conn = sqlite3.connect("users.db")
```

```
dummy_cursor = dummy_conn.cursor()
```

```
dummy_cursor.execute("DROP TABLE IF EXISTS  
users;")
```

```
dummy_cursor.execute("CREATE TABLE users  
(id INTEGER PRIMARY KEY, name TEXT, email  
TEXT);")
```

```
dummy_cursor.execute("INSERT INTO users (id,  
name, email) VALUES (1, 'Alice',  
'alice@example.com');")
```

```
dummy_cursor.execute("INSERT INTO users (id,
name, email) VALUES (2, 'Bob',
'bob@example.com');")

dummy_conn.commit()

dummy_conn.close()

user_input = input("Enter user ID: ")

print(get_user_data(user_input))
```

EXPLANATION:

Problem in Original	How It Was Fixed
SQL Injection Risk — Code was building query via <code>f"...{user_id}..."</code> which allows malicious input to become part of SQL syntax.	Switched to <i>parameterized query</i> using <code>?</code> placeholder: <code>cursor.execute(..., (user_id,))</code> .
Lack of Input Validation — <code>user_id</code> coming from input was used directly (as string) without checking its form.	Added checks: ensure it's an integer, positive. Converted via <code>int(input)</code> with try/except.
Resource Management — Simple <code>connect()</code> , <code>cursor()</code> , then manual <code>conn.close()</code> . If some exception occurs between, might leave resources open.	Use <code>with sqlite3.connect(...) as conn:</code> context manager; use <code>cursor()</code> inside; ensures connection (commit/close) is handled properly even on error.
Error Handling — In original, any database error would crash the program or propagate unexpected tracebacks. Also, calling <code>get_user_data(user_input)</code> where <code>user_input</code> is string would cause type issues.	Added <code>try/except</code> blocks: catch <code>sqlite3.Error</code> , wrap into <code>RuntimeError</code> , catch invalid input etc., log error messages.
Logging — No logging originally. Errors weren't recorded or traceable.	Added <code>logging</code> (info, error, debug) to help diagnose issues without exposing sensitive details to users.

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:

Generate python code and 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")).
```

CODE & OUTPUT:

The screenshot shows a Visual Studio Code editor window with a Python file named `calculate.py` open. The file contains a function `calculate(x, y, operation)` that performs arithmetic operations based on the `operation` parameter. It includes error handling for division by zero and unknown operations. A `main()` function demonstrates the usage of `calculate` with various inputs, including a division by zero case. The terminal at the bottom shows the command `python calculate.py` being executed, which results in an `Error: Cannot divide by zero` message.

```
1 def calculate(x, y, operation):
2     """
3     Calculate the result of an operation.
4     Parameters:
5     x (int): The first operand.
6     y (int): The second operand.
7     operation (str): The operation to perform ('+', '-', '*', '/').
8     Returns:
9     int: The result of the operation.
10    Raises:
11    ValueError: If the operation is unknown or if division by zero occurs.
12    """
13    if operation == '+':
14        return x + y
15    elif operation == '-':
16        return x - y
17    elif operation == '*':
18        return x * y
19    elif operation == '/':
20        if y == 0:
21            raise ValueError("Cannot divide by zero")
22        return x / y
23    else:
24        raise ValueError(f"Unknown operation: {operation}")
25
26 def main():
27     # Demonstration / simple usage
28     print("10 + 5 =", calculate(10, 5, "+"))
29     try:
30         print("10 ÷ 0 =", calculate(10, 0, "/"))
31     except Exception as e:
32         print("Error:", e)
33
34 if __name__ == "__main__":
35     main()
```

Terminal Output:

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/kadal/anaconda3/pkgs/alembic-1.16.4-py313haa95532_0/lib/site-packages/alembic/dl/TASK 5 AI LAB 10.3.py"
10 + 5 = 15
Error: Cannot divide by zero
PS C:\Users\kadal>
```

EXPLANATION:

Here's why and what was changed, and how each change addresses issues:

- **Function name** changed from `calc` to `calculate` to better express what it does.
- **Parameter name** `operation` instead of `z` — more intuitive.
- **Docstring** added: describes parameters, return value, and the kinds of errors that might be thrown. This helps others understand usage and edge cases.
- **Type checking**: ensure `x` and `y` are numeric types. If someone passes non-numeric, a `TypeError` is raised early.
- **Operation validation**: ensure `operation` is one of the expected set; if not, raise `ValueError`. No ambiguous or silent behavior.
- **Division by zero** is explicitly checked and handled (raises `ValueError`). Avoids unhandled `ZeroDivisionError` or crash.
- **Consistent return behavior**: function always returns a numeric result or raises error; does not perform any printing inside.
- **Main function** with example usage: calls to `calculate` wrapped in `try/except` to show how to handle possible exceptions.
- **Guard** `if __name__ == "__main__":` ensures sample code only runs when the script is run directly, not when imported.
- **Cleaner formatting**: 4 space indentation, spaces around operators and after commas, consistent structure in conditional branches.

