**ASSIGNMENT-10**

**NAME : G.SAI RAJ**

**HT NO: 2403A52149**

**BATCH:** 24BTCAIAIB06

**TASK1:**

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

# buggy\_code\_task1.py

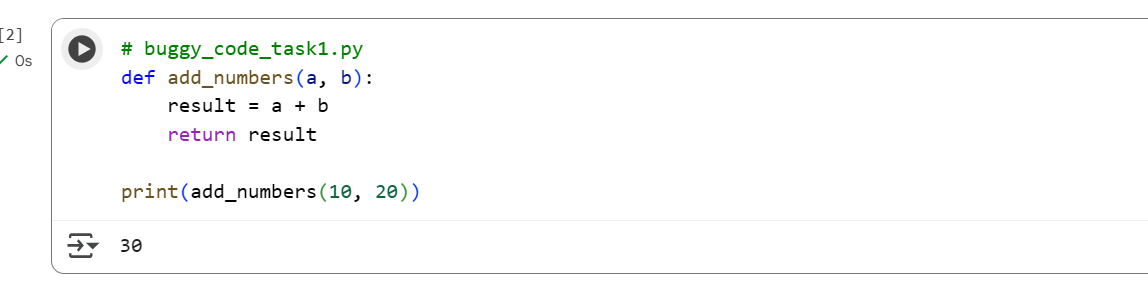
def add\_numbers(a, b)

result = a + b

return reslt

print(add\_numbers(10 20))

**CODE AND OUTPUT:**

****

**EXPLANATION:**

1. **Syntax Error (Missing Colon):** The original code had def add\_numbers(a, b), which was missing a colon at the end of the function definition. I added the colon to make it def add\_numbers(a, b):.
2. **Indentation Error:** The lines inside the function (result = a + b and return result) were not indented correctly under the function definition. I added the proper indentation (usually 4 spaces) to these lines so they are part of the function's body.
3. **Variable Error (Typo):** The original code had return reslt. There was a typo in the variable name, it should have been result. I corrected this to return result.
4. **Syntax Error (Missing Comma):** The original code had print(add\_numbers(10 20)), which was missing a comma between the arguments 10 and 20 when calling the function. I added the comma to make it print(add\_numbers(10, 20)).

These changes corrected the syntax, indentation, and variable errors, allowing the code to run without issues and produce the correct output

**TASK 2:**

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:

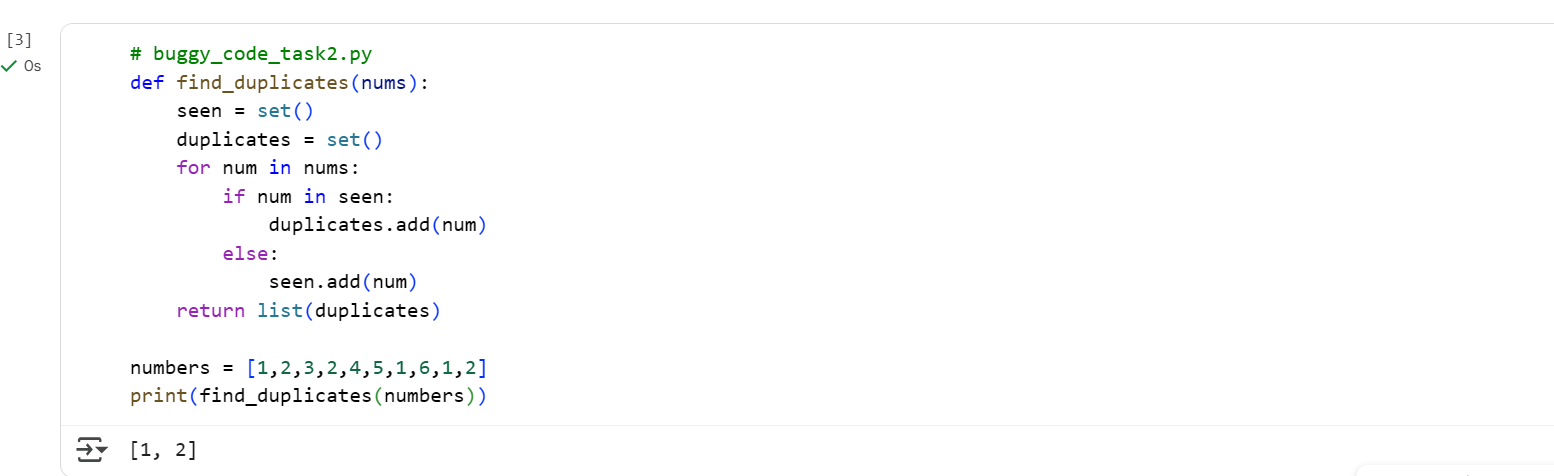
duplicates.append(nums[i])

return duplicates

numbers = [1,2,3,2,4,5,1,6,1,2]

print(find\_duplicates(numbers))

**CODE AND OUTPUT:**

****

**EXPLANATION:**

* **seen = set()**: This line initializes an empty set called seen. This set will be used to keep track of the numbers we have encountered so far in the list. Sets are very efficient for checking if an element is already present (num in seen).
* **duplicates = set()**: This line initializes another empty set called duplicates. This set will store the numbers that are found to be duplicates. Using a set here automatically handles adding each duplicate only once, even if it appears multiple times in the input list.
* **for num in nums:**: This loop iterates through each number (num) in the input list nums.
* **if num in seen:**: Inside the loop, this condition checks if the current number num is already present in the seen set.
* **duplicates.add(num)**: If the number is in the seen set, it means we have encountered this number before, so it's a duplicate. We add this duplicate number to the duplicates set.
* **else: seen.add(num)**: If the number is not in the seen set, it means this is the first time we are seeing this number. We add it to the seen set so we can detect it as a duplicate if we encounter it again later in the list.
* **return list(duplicates)**: Finally, after iterating through all the numbers in the input list, the function converts the duplicates set into a list and returns it.

This approach is more efficient than using nested loops because checking for membership in a set (in) takes on average constant time (O(1)), whereas iterating through a list to check for membership takes linear time (O(n)).

0 / 2000

Gemini can make mistakes so double-check it and use code with caution. [Learn more](http://g.co/legal/generative-code" \t "_blank)

**TASK 3:**

Prompt:

Generate a Function renamed to calculate\_factorial.

Proper indentation, variable naming, docstrings, and formatting.

Provide a more readable version.

def c(n):

x=1

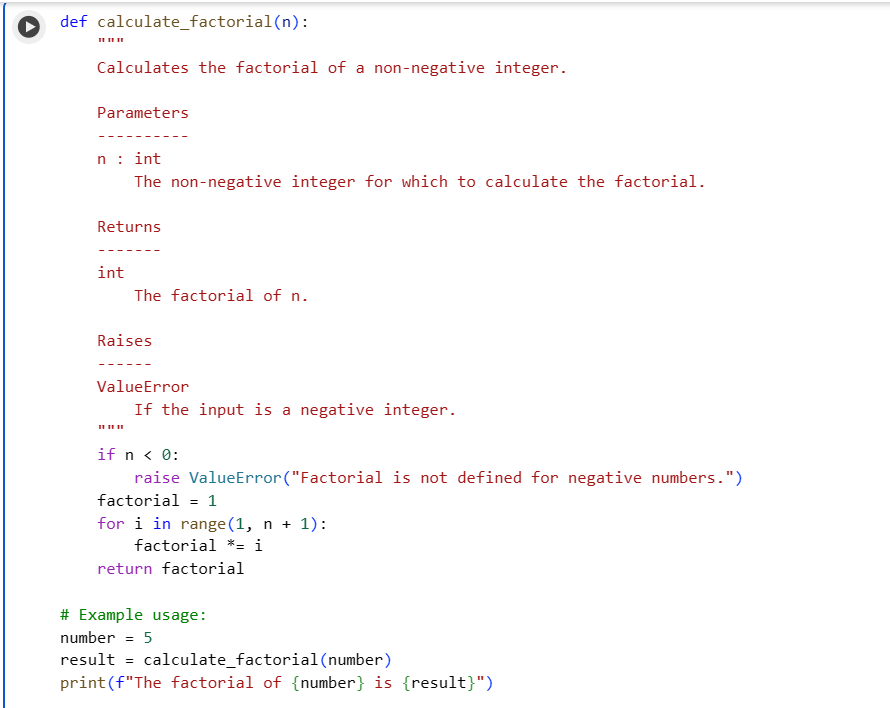
for i in range(1,n+1):

x=x\*i

return x

print(c(5))

**CODE AND OUPUT:**

****

1. **EXPLANATION:  
   def calculate\_factorial(n):**: This line defines the function calculate\_factorial that takes one argument, n.
2. **Docstring**: The triple-quoted string explains what the function does, its parameters (n), what it returns, and potential errors it might raise (ValueError). This is written in NumPy style, which is a common convention for documenting Python code.
3. **if n < 0:**: This checks if the input number n is negative.
4. **raise ValueError(...)**: If n is negative, a ValueError is raised because the factorial is not defined for negative numbers.
5. **factorial = 1**: Initializes a variable factorial to 1. This is the base case for the factorial calculation (0! and 1! are both 1).
6. **for i in range(1, n + 1):**: This loop iterates from 1 up to and including n.
7. **factorial \*= i**: In each iteration, the current value of factorial is multiplied by the loop variable i. This performs the cumulative multiplication needed for the factorial.
8. **return factorial**: After the loop finishes, the calculated factorial value is returned.
9. **Example Usage**: The code then demonstrates how to use the function with a positive number (number = 5) and prints the result.
10. **Error Handling Example**: A try...except block is included to show how the ValueError is handled when a negative number is passed to the function.

In essence, the function calculates n \* (n-1) \* (n-2) \* ... \* 1 for a non-negative integer n.

**TASK-4:**

Prompt:

Add security practices and exception handling to the code.

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()

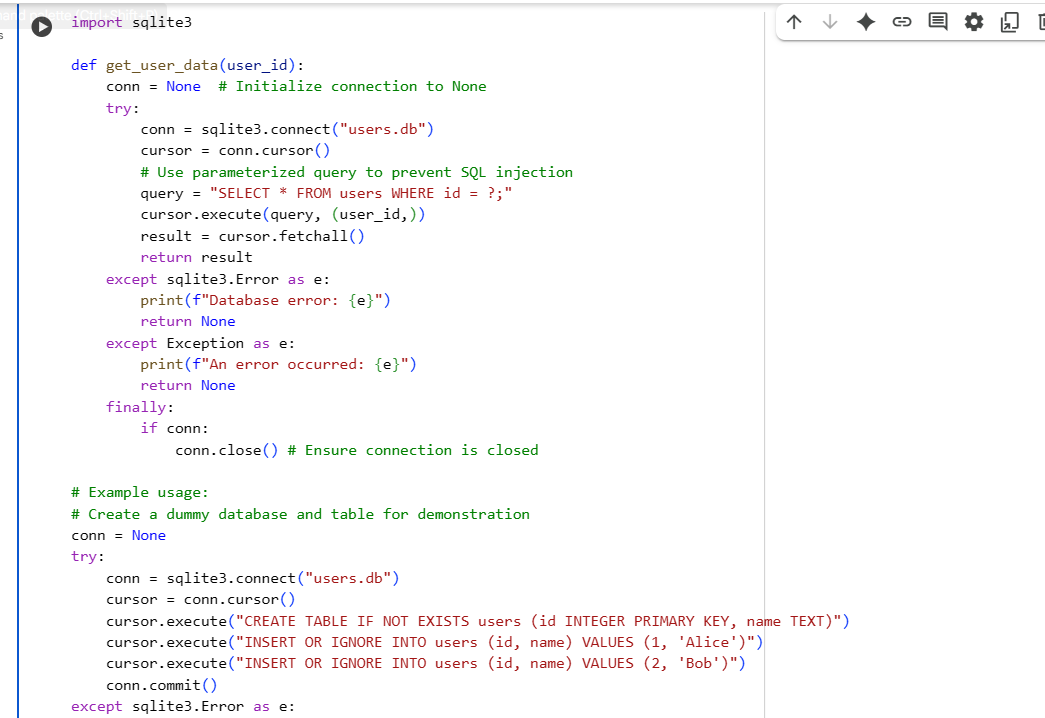
conn.close()

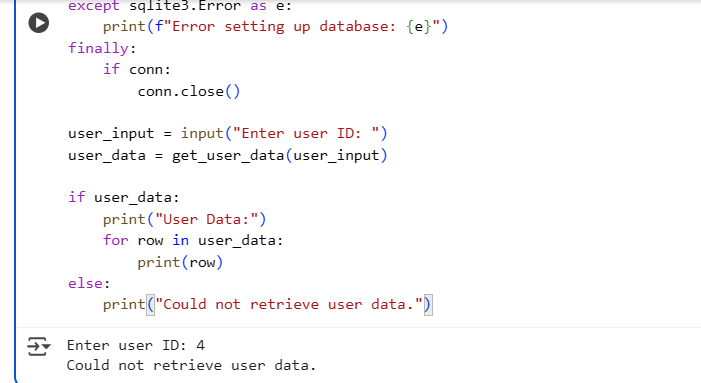
return result

user\_input = input("Enter user ID: ")

print(get\_user\_data(user\_input))

**CODE AND OUTPUT:**

****

****

**EXPLANATION:**

1. **import sqlite3**: This line imports the necessary library for working with SQLite databases in Python.
2. **def get\_user\_data(user\_id):**: This defines a function get\_user\_data that takes user\_id as input.
3. **conn = None**: Initializes a variable conn to None to hold the database connection.
4. **try...except...finally block**: This block is used for error handling and ensuring resources are properly managed.
   * **conn = sqlite3.connect("users.db")**: Attempts to establish a connection to the users.db database. If the database doesn't exist, it will be created.
   * **cursor = conn.cursor()**: Creates a cursor object, which is used to execute SQL commands.
   * **query = "SELECT \* FROM users WHERE id = ?;"**: Defines the SQL query to select all columns from the users table where the id matches. The ? is a placeholder for the user\_id, which is a security practice to prevent SQL injection.
   * **cursor.execute(query, (user\_id,))**: Executes the SQL query. The user\_id is passed as a tuple (user\_id,) to substitute the placeholder ?. This is the secure way to pass external values into a SQL query.
   * **result = cursor.fetchall()**: Fetches all rows from the executed query result.
   * **return result**: Returns the fetched data.
   * **except sqlite3.Error as e:**: Catches any errors specifically related to SQLite operations.
   * **except Exception as e:**: Catches any other general exceptions that might occur.
   * **finally:**: This block always executes, regardless of whether an exception occurred.
   * **if conn: conn.close()**: Checks if the conn variable has a valid connection and closes it to release the database resource.
5. **Example Usage**:
   * The code first sets up a dummy users table and inserts some sample data (Alice and Bob) for demonstration purposes. It uses CREATE TABLE IF NOT EXISTS and INSERT OR IGNORE to avoid errors if the table or data already exist.
   * **user\_input = input("Enter user ID: ")**: Prompts the user to enter a user ID.
   * **user\_data = get\_user\_data(user\_input)**: Calls the get\_user\_data function with the user's input.
   * **if user\_data: ... else: ...**: Checks if data was retrieved and prints it or a message indicating that data could not be retrieved.

In summary, this script provides a safe and robust way to query a user database by handling potential errors and preventing common security vulnerabilities like SQL injection.

**TASK-5:**

Prompt:

Generate a review report for this messy code.

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":

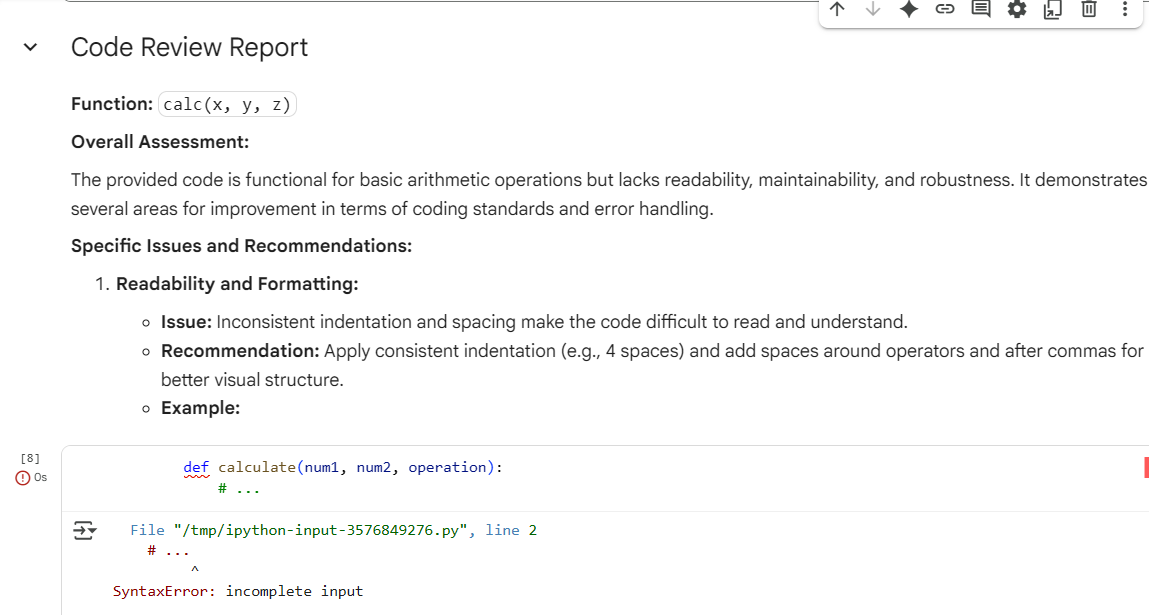
return x/y

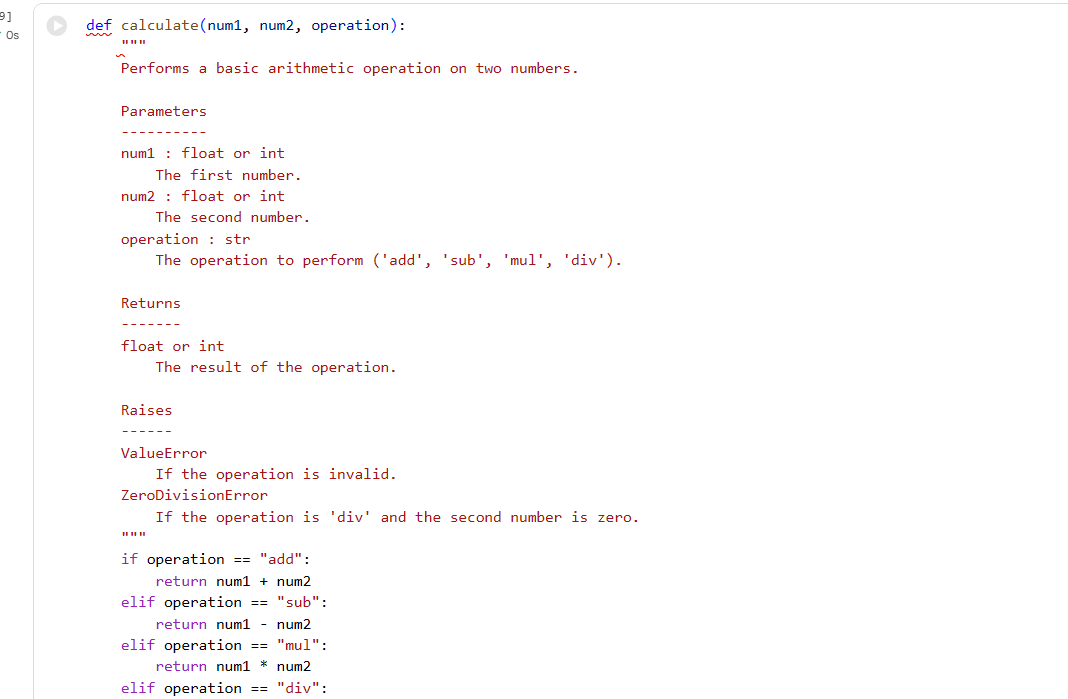
else: print("wrong")

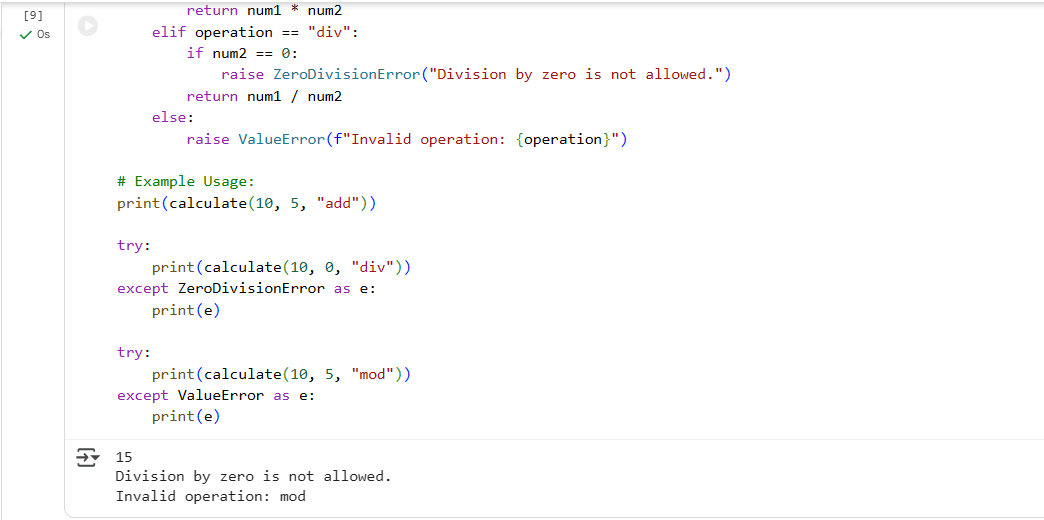
print(calc(10,5,"add"))

print(calc(10,0,"div"))

**CODE AND OUPUT:**

****

****

****