ASSIGNMENT-10

SYNTAX AND ERROR DECTECTION

HTNO:2403A51319

BRANCH:13

NAME:ANKITHA

**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

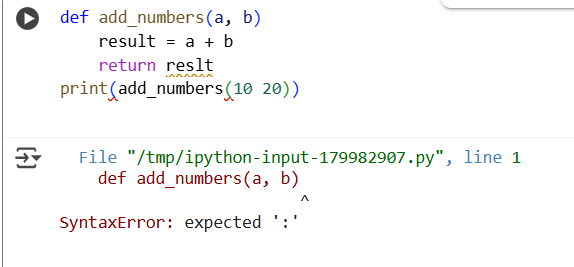
return reslt

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.
* PROMPT:
* Please correct the error

ERROR CODE:



CORRECT CODE:

A screenshot of a computer code

AI-generated content may be incorrect.

PROBLEMS IN THE CODE:

1. **Missing Colon:** The function definition def add\_numbers(a, b) was missing a colon : at the end. In Python, function definitions must end with a colon.
2. **Typo:** The variable name reslt in the return statement was misspelled. It should be result to match the variable where you stored the sum.
3. **Missing Comma:** When calling the function print(add\_numbers(10 20)), the arguments 10 and 20 were not separated by a comma. Function arguments in Python calls must be separated by commas.

EXPLANATION:

 defines a function called add\_numbers that takes two arguments, a and b. Inside the function, it calculates the sum of a and b and stores it in a variable called result. Then, it returns the value of result. Finally, the code calls the add\_numbers function with the values 10 and 20, and prints the returned result to the console.

**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))

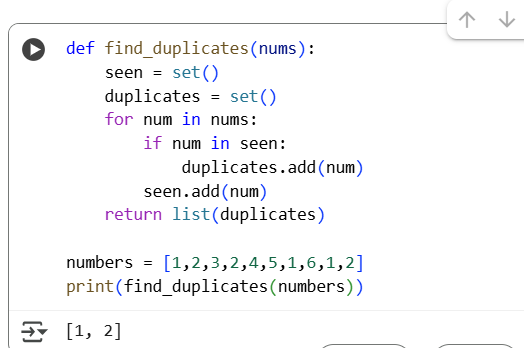
**Expected Output**:

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

PROMPT:

Decrease the code and loops

CODE:



PROBLEMS IN THIS CODE:

I see the code you've provided has a couple of syntax errors. I can fix them and explain what was wrong.

EXPLANATION:

1. **def find\_duplicates(nums):**: This line defines a function named find\_duplicates that accepts one argument, nums.
2. **seen = set()**: This line initializes an empty set called seen. This set will be used to keep track of the numbers encountered so far in the input list. Sets are used because checking for membership (in) and adding elements (add) are very efficient (average O(1) time complexity).
3. **duplicates = set()**: This line initializes another empty set called duplicates. This set will store the numbers that are found to be duplicates.
4. **for num in nums:**: This line starts a loop that iterates through each number (num) in the input list nums.
5. **if num in seen:**: Inside the loop, this line checks if the current number num is already present in the seen set.
6. **duplicates.add(num)**: If the if condition is true (meaning the number num has been seen before), this line adds num to the duplicates set.
7. **seen.add(num)**: After checking for duplicates, this line adds the current number num to the seen set. This ensures that the next time this number is encountered, it will be identified as a duplicate.
8. **return list(duplicates)**: After the loop finishes processing all the numbers in the input list, this line converts the duplicates set into a list and returns it.

The code then demonstrates the usage of this function with a sample list numbers = [1, 2, 3, 2, 4, 5, 1, 6, 1, 2] and prints the result, which is [1, 2] (the duplicate numbers in the list).

**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.

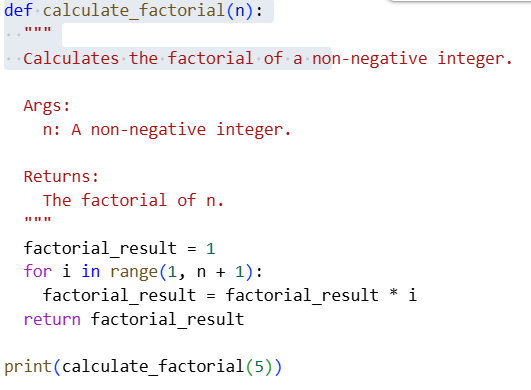
Proper indentation, variable naming, docstrings, and formatting.

AI should provide a more readable version.

PROMPT:

Rename the function c to calculate\_factorial with proper indentation,variable naming,and formatting.

CODE:



OUTPUT:

120

PROBLEM IN THIS CODE:

The code you provided to calculate the factorial is functionally correct, but it could be more readable. The variable and function names are not very descriptive, and it lacks a docstring. I will show you how to improve its readability.

EXPLANATION:

1. **def calculate\_factorial(n):**: This line defines a function named calculate\_factorial that takes one argument, n.
2. **""" ... """**: This is a docstring, which explains what the function does, its arguments (Args), and what it returns (Returns). It's good practice to include docstrings for your functions to make them easier to understand.
3. **factorial\_result = 1**: This line initializes a variable called factorial\_result to 1. This variable will store the calculated factorial. We start with 1 because the factorial of 0 is 1, and it's the multiplicative identity.
4. **for i in range(1, n + 1):**: This line starts a for loop that iterates from 1 up to and including the value of n.
5. **factorial\_result = factorial\_result \* i**: Inside the loop, this line multiplies the current value of factorial\_result by the loop variable i and updates factorial\_result with the new value. This is the core of the factorial calculation, where we multiply the result by each integer from 1 to n.
6. **return factorial\_result**: After the loop finishes, this line returns the final calculated value of factorial\_result, which is the factorial of n.

The code then calls the calculate\_factorial function with the argument 5 and prints the returned value to the console. The factorial of 5 is 5 \* 4 \* 3 \* 2 \* 1 = 120, so the output will be 120.

**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.

Prompt:

Do it in sql , Try-except block for database errors.

CODE:

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

EXPLANATION:

1. **import sqlite3**: This line imports the sqlite3 module, which provides an interface for working with SQLite databases in Python.
2. **Database Setup (outside the function)**:
   * **try...except sqlite3.Error as e:**: This block handles potential errors during the database setup process.
   * **conn\_setup = sqlite3.connect("users.db")**: This establishes a connection to the SQLite database file named "users.db". If the file doesn't exist, it will be created.
   * **cursor\_setup = conn\_setup.cursor()**: This creates a cursor object, which is used to execute SQL commands.
   * **cursor\_setup.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, email TEXT)")**: This executes an SQL command to create a table named users if it doesn't already exist. The table has three columns: id (an integer, which is the primary key), name (text), and email (text).
   * **cursor\_setup.execute("INSERT OR IGNORE INTO users (id, name, email) VALUES (1, 'Alice', 'alice@example.com')")** and **cursor\_setup.execute("INSERT OR IGNORE INTO users (id, name, email) VALUES (2, 'Bob', 'bob@example.com')")**: These lines execute SQL commands to insert two rows of data into the users table. INSERT OR IGNORE prevents errors if these rows already exist.
   * **conn\_setup.commit()**: This saves the changes made to the database.
   * **conn\_setup.close()**: This closes the connection to the database.
3. **def get\_user\_data(user\_id):**: This defines the function get\_user\_data that takes user\_id as an argument.
4. **Docstring**: Explains what the function does, its arguments, and what it returns.
5. **conn = None**: Initializes a variable conn to None. This is used later in the finally block to ensure the connection is closed even if an error occurs.
6. **try...except...finally block**: This structure is used for error handling and resource management.
   * **Input Validation (try...except ValueError)**:
     + **user\_id = int(user\_id)**: Attempts to convert the input user\_id to an integer.
     + **except ValueError:**: If the conversion fails (meaning the input is not a valid number), it prints an error message and returns None.
   * **Database Operations (try block)**:
     + **conn = sqlite3.connect("users.db")**: Establishes a connection to the "users.db" database.
     + **cursor = conn.cursor()**: Creates a cursor object.
     + **query = "SELECT \* FROM users WHERE id = ?;"**: Defines the SQL query to select all columns from the users table where the id matches a placeholder ?. **This is crucial for preventing SQL injection.**
     + **cursor.execute(query, (user\_id,))**: Executes the SQL query. The user\_id is passed as a tuple (user\_id,) to the execute method. SQLite handles substituting the placeholder ? with the actual user\_id value, ensuring the input is treated as data, not executable SQL code.
     + **result = cursor.fetchall()**: Fetches all the rows that match the query and returns them as a list of tuples.
     + **return result**: Returns the fetched data.
   * **except sqlite3.Error as e:**: Catches specific SQLite database errors, prints an error message, and returns None.
   * **except Exception as e:**: Catches any other unexpected errors, prints an error message, and returns None.
   * **finally:**: This block always executes, whether an error occurred or not.
     + **if conn:**: Checks if the conn variable is not None (meaning a connection was successfully established).
     + **conn.close()**: Closes the database connection. This is important to release resources.
7. **User Input and Function Call**:
   * **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.
8. **Displaying Results**:
   * **if user\_data:**: Checks if the user\_data variable is not None (meaning the function executed without major errors).
   * **if user\_data:**: This inner if condition is redundant, but it checks if the user\_data list is not empty (meaning a user was found).
   * **print("User data found:")**: Prints a message indicating that data was found.
   * **for row in user\_data:**: Iterates through the rows (tuples) returned by the function.
   * **print(row)**: Prints each row of user data.
   * **else: print("User not found.")**: If the user\_data list is empty, it prints "User not found."

**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
  + PROMPT:
  + Find the missing docstrings,inconsistent formatting,missing error handling for division by zero,non descriptive function.

CODE:

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

EXPLANATION:

1. **def calculate\_operation(x, y, operation):**: This line defines a function named calculate\_operation that takes three arguments: x (the first number), y (the second number), and operation (a string specifying the operation).
2. **""" ... """**: This is a docstring, which explains what the function does, its arguments (Args), and what it returns (Returns). It also mentions the possible values for the operation argument and the conditions under which it might return None.
3. **if operation == "add":**: This is the start of a series of if-elif-else statements that check the value of the operation argument. If operation is equal to the string "add", the code inside this block is executed.
   * **return x + y**: If the operation is "add", the function returns the sum of x and y.
4. **elif operation == "sub":**: If the previous if condition is false, this elif condition checks if operation is equal to the string "sub".
   * **return x - y**: If the operation is "sub", the function returns the difference between x and y.
5. **elif operation == "mul":**: If the previous conditions are false, this elif condition checks if operation is equal to the string "mul".
   * **return x \* y**: If the operation is "mul", the function returns the product of x and y.
6. **elif operation == "div":**: If the previous conditions are false, this elif condition checks if operation is equal to the string "div".
   * **if y == 0:**: Inside the division block, this checks if the second number y is zero.
     + **print("Error: Division by zero is not allowed.")**: If y is zero, it prints an error message because division by zero is undefined.
     + **return None**: It then returns None to indicate that the operation could not be completed due to division by zero.
   * **return x / y**: If y is not zero, the function returns the result of dividing x by y.
7. **else:**: If none of the previous if or elif conditions are true (meaning the operation string does not match "add", "sub", "mul", or "div").
   * **print("Error: Invalid operation.")**: It prints an error message indicating that the provided operation is invalid.
   * **return None**: It returns None to indicate that the operation could not be performed.
8. **Print Statements (Examples)**: The lines following the function definition demonstrate how to use the calculate\_operation function with different inputs and print the results. These examples show successful operations, division by zero error handling, and invalid operation handling.