

### Task 1: Syntax and Error Detection

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

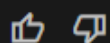
```
# buggy_code_task1.py
def add_numbers(a, b)
    result = a + b
    return reslt
print(add_numbers(10 20))
```

#### Expected Output:

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

**G** identify and fix the error and explain what have you done using comments

✦ I can help with that! It looks like there's a syntax error in the function definition and a typo in the variable name within the function. I'll fix those and add comments to explain.



▶ Accept & Run    ✓ Accept    ✕ Cancel

✦ What can I help you build?



Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

◆ Gemini



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

+# Fix: Added colon at the end of function definition
+def add_numbers(a, b):
+    # Fix: Corrected variable name typo from 'reslt' to 'result'
+    result = a + b
+    return result
+# Fix: Added comma between arguments in the print statement
+print(add_numbers(10, 20))
```



30



identify and fix the

## 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

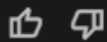
numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates(numbers))
```

### Expected Output:

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

**G** optimize the above code

✦ I can optimize the `find_duplicates` function for you. The current nested loop approach has a time complexity of  $O(n^2)$ . I can improve this by using a set to achieve a time complexity of  $O(n)$ .



▶ Accept & Run    ✓ Accept    ✕ Cancel

✦ What can I help you build?



Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

✦ Gemini

```
# buggy_code_task2.py
+## Optimized function to find duplicates using a set
def find_duplicates(nums):
-     duplicates = []
+     seen = set() # Use a set to store seen elements
+     duplicates = []
+     for num in nums:
+         if num in seen:
+             # If the number is already in the set, it's a duplicate
+             if num not in duplicates: # Add to duplicates only if not already added
+                 duplicates.append(num)
-         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])
+         else:
+             # If the number is not in the set, add it
+             seen.add(num)
-     return duplicates
+     return duplicates
+
numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates(numbers))
```

🔄 [2, 1]

### 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.

refactor the above code PEP-8 compliant and more understandable

I can help you refactor the code in cell `cCbAqnIEWTny` to be more PEP 8 compliant and understandable. I will correct the indentation, rename the function to be more descriptive, and add comments.

▶ Accept & Run    ✓ Accept    ✕ Cancel

What can I help you build?



ni can make mistakes so double-check it and use code with caution. [Learn more](#)

```
# buggy_code_task3.py
def c(n):
    x=1
    for i in range(1,n+1):
        x=x*i
    return x
print(c(5))

+ # Refactored function to calculate factorial
+ def calculate_factorial(n): # Renamed function for clarity and added colon
+     # Initialize the factorial result
+     result = 1
+     # Loop from 1 to n (inclusive) to calculate the factorial
+     for i in range(1, n + 1):
+         # Multiply the result by the current number
+         result = result * i
+     # Return the calculated factorial
+     return result
+
+ # Call the function with n=5 and print the result
+ print(calculate_factorial(5))
```

↔ 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.

Add security practices and exception handling to the code.

```

# Gemini
# 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

+ conn = None # Initialize connection to None
+ try:
+     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,))
+     result = cursor.fetchall()
+     return result
+ except sqlite3.Error as e:
+     print(f"Database error: {e}")
+     return None
+ except Exception as e:
+     print(f"An error occurred: {e}")
+     return None
+ finally:
+     if conn: # Close connection only if it was successfully opened
+         conn.close()

+
+ user_input = input("Enter user ID: ")
- print(get_user_data(user_input))
+
+ # Add validation for user input
+ try:
+     user_id = int(user_input) # Attempt to convert input to integer
+     user_data = get_user_data(user_id)
+     if user_data:
+         print("User Data:", user_data)
+     else:
+         print("could not retrieve user data.")
+ except ValueError:
+     print("Invalid input. Please enter a numeric user ID.")

Enter user ID: 101
Database error: no such table: users
Could not retrieve user data.
```

## 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

give a review report for the above code which should contain Missing docstrings, Inconsistent formatting, Missing error handling for division by zero, Non-descriptive function/variable names, Suggestions for readability and PEP 8 compliance



```
Gemini
# 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"))
+
+# Function to perform basic arithmetic operations
+def perform_arithmetic_operation(operand1, operand2, operator):
+    if operator == "add":
+        return operand1 + operand2
+    elif operator == "sub":
+        return operand1 - operand2
+    elif operator == "mul":
+        return operand1 * operand2
+    elif operator == "div":
+        # Handle division by zero
+        if operand2 == 0:
+            print("Error: Division by zero is not allowed.")
+            return None # Return None or raise an exception
+        return operand1 / operand2
+    else:
+        print("Error: Invalid operator.")
+        return None # Return None or raise an exception
+
+# Call the function with different operations and print the results
+print(perform_arithmetic_operation(10, 5, "add"))
+print(perform_arithmetic_operation(10, 5, "add"))
+# Expected output: 15
+
+print(perform_arithmetic_operation(10, 0, "div"))
+# Expected output: Error message and None

15
Error: Division by zero is not allowed.
None
```

```
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"))
```

#### Issues Identified:

- Missing Docstrings:** The function `calc` lacks a docstring. Docstrings are essential for explaining what a function does, its parameters, and what it returns. This makes the code harder to understand and maintain.
- Inconsistent Formatting:** The code has inconsistent indentation and spacing. For example, the `if` and `elif` statements have varying indentation levels, and there's no consistent spacing around operators and keywords. This violates PEP 8 guidelines and makes the code visually messy and harder to read.
- Missing Error Handling for Division by Zero:** The code does not handle the case of division by zero. When `z` is "div" and `y` is 0, a `ZeroDivisionError` will occur, crashing the program.
- Non-Descriptive Function/Variable Names:**
  - The function name `calc` is too short and not descriptive of its purpose. A more informative name would be `perform_calculation` or `arithmetic_operation`.
  - The variable names `x`, `y`, and `z` are also not descriptive. It's unclear what `x` and `y` represent (e.g., operand1, operand2) and what `z` represents (e.g., operation, operator).

- Lack of Clear Return Value for Invalid Operation:** When `z` does not match any of the supported operations, the code prints "wrong" but doesn't return a specific value. It implicitly returns `None`. It would be better to raise an error or return a specific value (like `None`) and handle this case explicitly where the function is called.

#### Suggestions for Readability and PEP 8 Compliance:

- Add a Docstring:** Add a docstring to the `calc` function explaining its purpose, arguments, and return value.
- Consistent Indentation and Spacing:** Apply consistent 4-space indentation and follow PEP 8 guidelines for spacing around operators, keywords, and function definitions.
- Implement Error Handling for Division by Zero:** Add a check within the `elif z=="div":` block to see if `y` is zero. If it is, raise a `ZeroDivisionError` or return an appropriate value (like `None`) and handle it outside the function.
- Use Descriptive Names:** Rename the function `calc` to something more descriptive like `perform_arithmetic_operation`. Rename the variables `x`, `y`, and `z` to more meaningful names like `operand1`, `operand2`, and `operator`.
- Improve Handling of Invalid Operations:** Instead of just printing "wrong", consider raising a `ValueError` for an invalid operator or returning a specific value like `None` and checking for it when calling the function.
- Use a Dictionary for Operations (Optional but Recommended):** For better readability and extensibility, you could use a dictionary to map operator strings to the corresponding arithmetic operations.