AI ASSISTED CODING
LAB ASS 10.4

TASK 1

---

def add_numbers(a, b):

  result = a + b

  return result

print(add_numbers(10, 20))

OUTPUT :

```
⮑ 30
```

EXPLANATION :

1. **def add_numbers(a, b)::** This line defines the function named add_numbers and specifies that it accepts two parameters, a and b. The colon : indicates the start of the function's code block.

2. **result = a + b**: Inside the function, this line calculates the sum of the two input parameters a and b and stores the result in a variable named result.

3. **return result**: This line returns the value stored in the result variable back to the part of the code that called the function.

4. **print(add_numbers(10, 20))**: This line calls the add_numbers function with the arguments 10 and 20. The function executes, calculates the sum (30), and returns it. The print() function then displays the returned value (30) to the console.

```python
def find_duplicates_optimized(nums):

    seen = set()

    duplicates = set()

    for num in nums:

        if num in seen:

            duplicates.add(num)

        seen.add(num)

    return list(duplicates)


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

print(find_duplicates_optimized(numbers))
```

OUT PUT :

```
[1, 2]
```

The original code used nested loops, resulting in a time complexity of O(n^2), which becomes very slow for larger lists.

The optimized code uses two sets: seen and duplicates.

1. **seen = set(): This set stores all the numbers encountered so far.**

2. **duplicates = set(): This set stores the unique duplicate numbers found.**

3. **for num in nums:: We iterate through the input list nums once.**

4. **if num in seen:: For each number, we check if it's already in the seen set. If it is, it means we've seen this number before, so it's a duplicate.**

5. **duplicates.add(num): If the number is a duplicate, we add it to the duplicates set. Since sets only store unique elements, adding a duplicate that's already in the duplicates set has no effect.**

6. **seen.add(num): Regardless of whether the number is a duplicate or not, we add it to the seen set so that we can track which numbers we've encountered.**

7. **return list(duplicates):** Finally, we convert the duplicates set back to a list and return it.

This optimized approach has a time complexity of O(n) because we iterate through the list only once, and set operations (checking for membership and adding elements) have an average time complexity of O(1). This is a significant improvement over the O(n^2) complexity of the original code, especially for large input lists.

Here are the fixes I made to the code:

- **Syntax error: Added a colon : after the function definition def add_numbers(a, b).**

- **Variable error: Corrected the variable name reslt to result in the return statement.**

- **Syntax error: Added a comma , between the arguments 10 and 20 in the function call print(add_numbers(10 20)).**

TASK 3

```
def calculate_factorial(number):
    """
    Calculates the factorial of a non-negative integer.

    Args:
        number: A non-negative integer.

    Returns:
        The factorial of the input number.
    """
    result = 1
```

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

*result = result * i*

return result

print(calculate_factorial(5))

OUTPUT :

⇥ 120

EXPLANATION :

- **Function Renaming:** The function name was changed from c to calculate_factorial to be more descriptive of its purpose.

- **Improved Variable Names:** The variable x was renamed to result and n was renamed to number for better clarity.

- **Docstring Added:** A docstring was added to explain what the function does, its arguments, and what it returns. This is good practice for code documentation.

- **PEP 8 Indentation:** The indentation has been corrected to follow PEP 8 standards, using 4 spaces for each level of indentation.

- **PEP 8 Spacing:** Added spaces around operators (=, *, +) and after the comma in the range() function for better readability.

Here's the optimized code and the explanation of the optimization:

The original code used nested loops, resulting in a time complexity of O(n^2), which becomes very slow for larger lists.

The optimized code uses two sets: seen and duplicates.

1. **seen = set()**: This set stores all the numbers encountered so far.

2. **duplicates = set()**: This set stores the unique duplicate numbers found.

3. **for num in nums:**: We iterate through the input list nums once.

4. **if num in seen:**: For each number, we check if it's already in the seen set. If it is, it means we've seen this number before, so it's a duplicate.

5. **duplicates.add(num)**: If the number is a duplicate, we add it to the duplicates set. Since sets only store unique elements, adding a duplicate that's already in the duplicates set has no effect.

6. **seen.add(num)**: Regardless of whether the number is a duplicate or not, we add it to the seen set so that we can track which numbers we've encountered.

7. **return list(duplicates)**: Finally, we convert the duplicates set back to a list and return it.

This optimized approach has a time complexity of O(n) because we iterate through the list only once, and set operations (checking for membership and adding elements) have an average time complexity of O(1). This is a significant improvement over the O(n^2) complexity of the original code, especially for large input lists.

Here are the fixes I made to the code:

- **Syntax error:** Added a colon : after the function definition def add_numbers(a, b).

- **Variable error:** Corrected the variable name reslt to result in the return statement.

- **Syntax error:** Added a comma , between the arguments 10 and 20 in the function call print(add_numbers(10 20)).

TASK 4

import sqlite3

def get_user_data_enhanced(user_id):

    """

    Retrieves user data from the database with enhanced security and error handling.

    Args:

```python
        user_id: The ID of the user to retrieve.

    Returns:
        The user data if found, otherwise None.
    """
    conn = None  # Initialize connection to None
    try:
        # Input validation: Check if user_id is a digit
        if not str(user_id).isdigit():
            print("Invalid user ID. Please enter a numeric ID.")
            return None


        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 unexpected error occurred: {e}")
        return None
```

```python
        finally:

            if conn:

                conn.close()


# Example usage:

# Create a dummy database and table for demonstration

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

cursor = conn.cursor()

cursor.execute("DROP TABLE IF EXISTS users;")

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

cursor.execute("INSERT INTO users (name) VALUES ('Alice');")

cursor.execute("INSERT INTO users (name) VALUES ('Bob');")

conn.commit()

conn.close()


user_input = input("Enter user ID: ")

user_data = get_user_data_enhanced(user_input)


if user_data:

    print("User Data:", user_data)

else:

    print("User not found or invalid input.")
```

OUT PUT :

```
⮕  Enter user ID: 1
   User Data: [(1, 'Alice')]
```

EXPLANATION :

1. **Parameterized Query:**

- o **Change: Instead of directly embedding the user_id into the SQL query string using an f-string, we now use a placeholder ? in the query string: query = "SELECT * FROM users WHERE id = ?;".**

- o **Explanation: This is the standard and secure way to pass values to SQL queries in sqlite3. When you execute the query with the user_id as the second argument to cursor.execute(), the sqlite3 library handles the proper escaping of the value, preventing malicious input from altering the query structure (SQL injection).**

2. **Error Handling (Try-Except-Finally):**

   - o **Change: The database operations are wrapped in a try…except…finally block.**

   - o **Explanation:**

     - ▪ **The try block contains the code that might raise an exception (database connection, query execution).**

     - ▪ **The except sqlite3.Error as e: block catches specific errors related to the sqlite3 library and prints an informative error message.**

     - ▪ **The except Exception as e: block catches any other unexpected errors that might occur.**

     - ▪ **The finally block ensures that the database connection conn.close() is always executed, regardless of whether an error occurred or not. This is crucial for releasing resources.**

3. **Input Validation:**

   - o **Change: Added a check if not str(user_id).isdigit(): at the beginning of the function.**

   - o **Explanation: This basic validation checks if the provided user_id can be converted to a digit. While not exhaustive for all potential malicious inputs, it's a simple first line of defense to ensure the input is in an expected format before interacting with the database.**

4. **Connection Initialization:**

   - o **Change: conn = None is added before the try block.**

- o **Explanation: This ensures that conn is always defined, even if an error occurs during the connection attempt, preventing a NameError in the finally block.**

These changes significantly improve the security and robustness of your code when interacting with the database.

Here's the optimized code and the explanation of the optimization:

The original code used nested loops, resulting in a time complexity of O(n^2), which becomes very slow for larger lists.

The optimized code uses two sets: seen and duplicates.

1. **seen = set(): This set stores all the numbers encountered so far.**

2. **duplicates = set(): This set stores the unique duplicate numbers found.**

3. **for num in nums:: We iterate through the input list nums once.**

4. **if num in seen:: For each number, we check if it's already in the seen set. If it is, it means we've seen this number before, so it's a duplicate.**

5. **duplicates.add(num): If the number is a duplicate, we add it to the duplicates set. Since sets only store unique elements, adding a duplicate that's already in the duplicates set has no effect.**

6. **seen.add(num): Regardless of whether the number is a duplicate or not, we add it to the seen set so that we can track which numbers we've encountered.**

7. **return list(duplicates): Finally, we convert the duplicates set back to a list and return it.**

This optimized approach has a time complexity of O(n) because we iterate through the list only once, and set operations (checking for membership and adding elements) have an average time complexity of O(1). This is a significant improvement over the O(n^2) complexity of the original code, especially for large input lists.

Here are the fixes I made to the code:

- **Syntax error: Added a colon : after the function definition def add_numbers(a, b).**

- **Variable error: Corrected the variable name reslt to result in the return statement.**

- **Syntax error: Added a comma , between the arguments 10 and 20 in the function call print(add_numbers(10 20)).**

TASK 5

```python
import sys

def calculate(num1, num2, operation):
    """
    Performs basic arithmetic operations on two numbers.

    Args:
        num1: The first number.
        num2: The second number.
        operation: A string representing the operation ("add", "sub", "mul", "div").

    Returns:
        The result of the operation, or None if the operation is invalid or
        division by zero occurs.
    """
    if operation == "add":
        return num1 + num2
    elif operation == "sub":
        return num1 - num2
    elif operation == "mul":
        return num1 * num2
```

```python
    elif operation == "div":

        if num2 == 0:

            print("Error: Division by zero is not allowed.", file=sys.stderr)

            return None

        return num1 / num2

    else:

        print(f"Error: Invalid operation '{operation}'.", file=sys.stderr)

        return None


# Example usage:

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

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

print(calculate(10, 5, "mul"))

print(calculate(10, 5, "sub"))

print(calculate(10, 5, "mod"))
```

OUT PUT :

```
15
None
50
5
None
Error: Division by zero is not allowed.
Error: Invalid operation 'mod'.
```

EXPLANATION :

Here is the refactored code based on the code review report and an explanation of the changes:

1. **Descriptive Function and Variable Names:**

   o The function name is changed from calc to calculate.

   o Variable names x, y, and z are changed to num1, num2, and operation respectively.

2. **Added Docstring:**

- A docstring is added to explain the function's purpose, arguments, and return value.

3. **Improved Formatting and PEP 8 Compliance:**

    - Consistent 4-space indentation is used throughout the function.

    - Each return statement is on its own line.

    - Spaces are added around operators (+, -, *, /, ==).

    - A space is added after the comma in the function definition.

4. **Error Handling for Division by Zero:**

    - An if condition is added within the "div" block to check if num2 is zero.

    - If num2 is zero, an error message is printed to standard error (sys.stderr), and the function returns None to indicate an unsuccessful operation.

5. **Improved Handling of Invalid Operations:**

    - The else block now prints a more informative error message to standard error indicating the invalid operation.

    - The function returns None in the else block as well.

General Observations:

The code implements a basic calculator function. While it performs the intended operations, there are several areas where readability, maintainability, and robustness can be significantly improved.

Specific Issues and Recommendations:

1. **Missing Docstring:**

    - **Issue:** The function calc lacks a docstring.

    - **Recommendation:** Add a docstring to explain what the function does, its parameters (x, y, z), and what it returns. This is crucial for understanding the function's purpose without having to read the code itself.

2. **Inconsistent Formatting:**

    - **Issue:** The code has inconsistent indentation and uses an inline return statement (elif z=="sub": return x-y).

    - **Recommendation:** Adhere to PEP 8 guidelines for consistent indentation (use 4 spaces) and place each statement on a new line for better readability.

3. **Missing Error Handling for Division by Zero:**

   o **Issue:** The code does not handle the case where z is "div" and y is 0, which will result in a ZeroDivisionError.

   o **Recommendation:** Implement error handling (e.g., using a try-except block) to gracefully handle division by zero and provide an informative error message to the user instead of crashing the program.

4. **Non-Descriptive Function and Variable Names:**

   o **Issue:** The function name calc and variable names x, y, and z are not descriptive.

   o **Recommendation:** Use more meaningful names, such as calculate, num1, num2, and operation, to improve code clarity and make it easier to understand the code's purpose.

5. **Suggestions for Readability and PEP 8 Compliance:**

   o **Issue:** The code is not fully compliant with PEP 8.

   o **Recommendation:**

     ▪ Add spaces around operators (+, -, *, /, ==).

     ▪ Add a space after the comma in function calls and definitions.

     ▪ Ensure consistent vertical spacing between code blocks.