**Assignment-10.4**

Task-1:

Corrected code:

A black screen with white text

AI-generated content may be incorrect.

Output:

A screenshot of a computer

AI-generated content may be incorrect.

Explanation:

1. **def add\_numbers(a, b):**: This line defines the function add\_numbers. The def keyword is used to start a function definition. a and b are parameters, which are placeholders for the values that will be passed into the function when it is called. The colon : at the end of the line is required to indicate the start of the function's code block.
2. **result = a + b**: This line is inside the function. It calculates the sum of a and b and stores the result in a variable named result. This line is indented, which is how Python knows it's part of the add\_numbers function.
3. **return result**: This line is also inside the function and indented. The return keyword sends the value of the result variable back to wherever the function was called.
4. **print(add\_numbers(10, 20))**: This line is outside the function definition. It calls the add\_numbers function with the values 10 for a and 20 for b. The function will execute, calculate 10 + 20, which is 30, and return 30. The print() function then displays the returned value (30) to the console.

Task-2:

Code:

A screen shot of a computer

AI-generated content may be incorrect.

Output:

A black background with white text

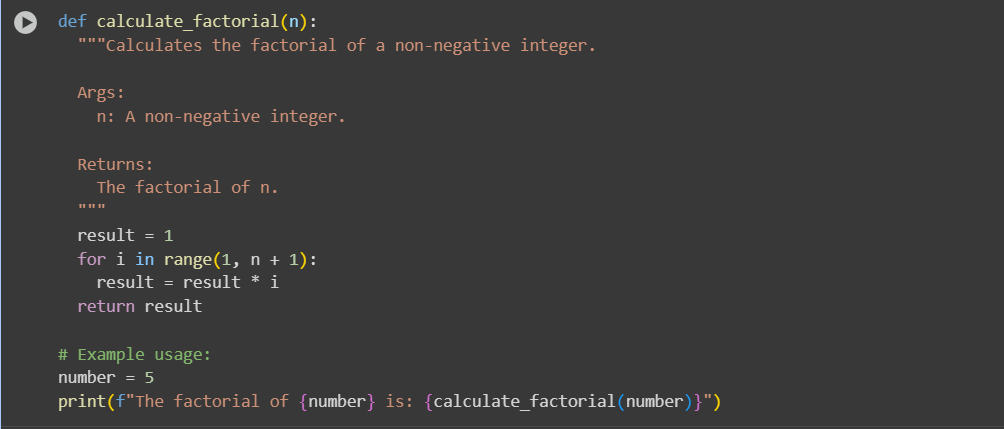
AI-generated content may be incorrect.

Explanation:

1. **def find\_duplicates\_optimized(nums):**: This line defines a function named find\_duplicates\_optimized that takes one argument, nums, which is expected to be a list of numbers.
2. **seen = set()**: Inside the function, an empty set called seen is created. This set will be used to store the unique numbers encountered so far as we iterate through the input list.
3. **duplicates = set()**: Another empty set called duplicates is created. This set will store the numbers that are found to be duplicates.
4. **for num in nums:**: This starts a loop that iterates through each element in the input list nums. In each iteration, the current element is assigned to the variable num.
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 condition in step 5 is true (meaning num is already in seen), it means we've found a duplicate. This line adds the duplicate number num to the duplicates set.
7. **else:**: If the condition in step 5 is false (meaning num is not in seen), it means this is the first time we're seeing this number.
8. **seen.add(num)**: This line adds the unique number num to the seen set so that we can track that we've encountered it.
9. **return list(duplicates)**: After the loop has finished iterating through all the numbers in nums, this line converts the duplicates set into a list and returns it. The reason we convert it to a list is that the original function returned a list, and it's good practice to maintain the expected output type.

Task-3:

Code:



Output:



Explanation:

1. **Function Definition:** The code first defines a function called calculate\_factorial that takes one argument, n. The docstring explains what the function does, its arguments, and what it returns.
2. **Initialization:** Inside the function, a variable result is initialized to 1. This is the base case for factorial calculation (0! and 1! are both 1).
3. **Looping through Numbers:** A for loop starts, iterating through the numbers from 1 up to and including the value of n (since range(1, n + 1) goes up to, but does not include, the stop value, we add 1 to include n).
4. **Calculating Factorial:** In each iteration of the loop, the current value of result is multiplied by the loop variable i, and the product is stored back in result. This repeatedly multiplies result by each integer from 1 to n.
5. **Returning the Result:** After the loop finishes, the function returns the final value stored in result, which is the factorial of n.
6. **Example Usage:** Outside the function, a variable number is set to 5.
7. **Function Call:** The calculate\_factorial function is called with number (which is 5) as the argument.
8. **Printing the Output:** The print() function then displays a formatted string using an f-string. This string includes the original number and the result returned by the calculate\_factorial(number) function call.

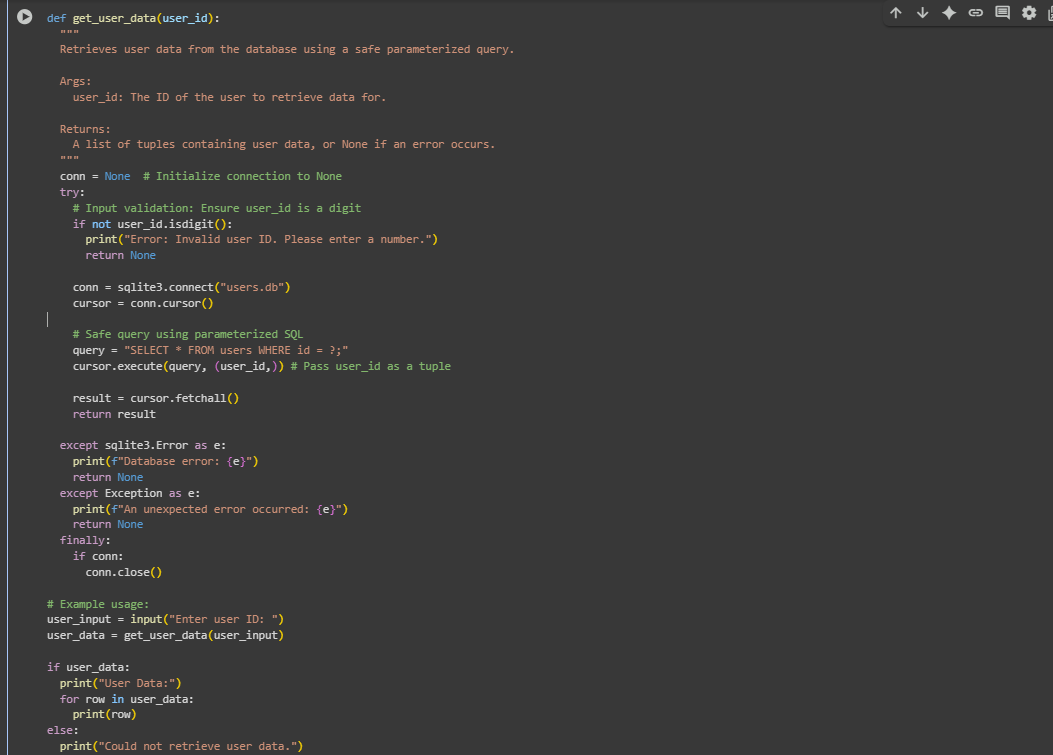
When this cell is executed with number = 5, the output will be:

The factorial of 5 is: 120

This is because the calculate\_factorial(5) call will calculate 5 \* 4 \* 3 \* 2 \* 1, which equals 120.

Task-4:

Code:



Output:

A screenshot of a computer

AI-generated content may be incorrect.

Explanation:

1. **Import sqlite3:** The code first imports the sqlite3 library, which is needed to interact with SQLite databases.
2. **Define get\_user\_data Function:** The get\_user\_data function is defined, which takes one argument, user\_id. It includes a docstring explaining its purpose, arguments, and return value.
3. **Initialize conn:** Inside the function, a variable conn is initialized to None. This variable will hold the database connection object. Initializing it to None is helpful in the finally block to check if the connection was successfully established before trying to close it.
4. **Start try Block:** A try block begins. This block contains the code that might potentially raise exceptions, such as database errors or issues with input validation.
5. **Input Validation:** Inside the try block, the code checks if the user\_id consists only of digits using user\_id.isdigit().
6. **Handle Invalid Input:** If user\_id.isdigit() is False, an error message is printed, the function returns None, and the rest of the try block is skipped.
7. **Establish Database Connection:** If the input is valid (only digits), the code attempts to connect to the "users.db" database using sqlite3.connect("users.db"). The connection object is assigned to the conn variable.
8. **Create Cursor:** A cursor object is created using conn.cursor(). The cursor is used to execute SQL commands.
9. **Define Parameterized Query:** The SQL query is defined using a placeholder ? for the user\_id. This is the secure way to include user input in a query and prevent SQL injection.
10. **Execute Query:** The cursor.execute(query, (user\_id,)) line executes the SQL query. The user\_id is passed as a tuple (user\_id,) as the second argument. sqlite3 handles substituting the ? with the actual value of user\_id in a safe manner.
11. **Fetch Results:** cursor.fetchall() retrieves all the rows returned by the query. The result is a list of tuples, where each tuple represents a row from the database. This result is stored in the result variable.
12. **Return Result:** The function returns the result (the list of tuples containing user data).
13. **Handle Database Errors:** If an sqlite3.Error occurs within the try block (e.g., the database file doesn't exist, or there's an issue with the SQL query), the code in the except sqlite3.Error as e: block is executed. It prints a "Database error:" message along with the specific error details and returns None.
14. **Handle Other Errors:** If any other unexpected Exception occurs within the try block, the code in the except Exception as e: block is executed. It prints an "An unexpected error occurred:" message along with the error details and returns None.
15. **finally Block (Close Connection):** The finally block is always executed, regardless of whether an exception occurred or not. It checks if the conn variable is not None (meaning a connection was successfully established). If it is not None, conn.close() is called to close the database connection and release resources.
16. **Get User Input:** Outside the function, input("Enter user ID: ") prompts the user to enter a user ID, and the input is stored in the user\_input variable.
17. **Call get\_user\_data:** The get\_user\_data function is called with the user\_input as the argument. The return value (either user data or None) is stored in the user\_data variable.
18. **Check and Print Data:** An if user\_data: statement checks if the user\_data variable contains a value (i.e., it's not None).
19. **Print User Data:** If user\_data is not None, the code prints "User Data:" and then iterates through the list of tuples in user\_data, printing each row.
20. **Handle No Data:** If user\_data is None (meaning there was an error or no data found for the given ID), the code in the else block is executed, printing "Could not retrieve user data."

Task-5

Code Review Report: buggy\_code\_task5.py

Code:

A screenshot of a computer program

AI-generated content may be incorrect.

Output:

A screenshot of a computer

AI-generated content may be incorrect.

EXPLANATION:

Code Review Report Analysis:

The original code had a few areas that could be improved for readability, maintainability, and robustness:

1. Poor Naming: The function name calc and the parameter names x, y, and z are not descriptive. It's hard to tell at a glance what the function does or what the parameters represent.

2. Inconsistent Formatting and Indentation: The indentation is inconsistent, especially in the elif z=="sub": return x-y line, which makes the code harder to read and understand the code structure. While Python is flexible with single-line statements, consistent indentation is crucial for readability and following PEP 8.

3. Lack of Error Handling for Division by Zero: The code does not handle the case where the operation is "div" and the second number (y) is zero. This would result in a ZeroDivisionError and crash the program.

4. Handling of Invalid Operations: For invalid operations, the code just prints "wrong". It would be better to provide a more informative error message and potentially return a specific value (like None) or raise an exception to indicate that the operation failed.

5. No Docstring: The function lacks a docstring, which is a good practice to explain what the function does, its parameters, and what it returns.

Improved Version Explanation:

The improved code addresses these points:

# Improved version (Illustrative)

def perform\_operation(num1, num2, operation):

"""Performs a basic arithmetic operation based on the provided string."""

if operation == "add":

return num1 + num2

elif operation == "sub":

return num1 - num2

elif operation == "mul":

return num1 \* num2

elif operation == "div":

if num2 == 0:

print("Error: Division by zero.")

return None # Or raise a ValueError

return num1 / num2

else:

print(f"Error: Invalid operation '{operation}'.")

return None # Or raise a ValueError

# Example usage:

print(perform\_operation(10, 5, "add"))

print(perform\_operation(10, 0, "div"))

print(perform\_operation(10, 5, "mod")) # Example of invalid operation

1. Clearer Naming: The function is renamed to perform\_operation, and parameters are named num1, num2, and operation. This makes the code's purpose and parameters much clearer.

2. Consistent Indentation: The indentation is consistent throughout the function, following standard Python practices and PEP 8.

3. Division by Zero Handling: An if num2 == 0: check is added within the "div" block. If num2 is 0, it prints an error message and returns None (or you could raise a ValueError for more explicit error handling).

4. Improved Invalid Operation Handling: The else block now prints a more informative error message using an f-string, indicating which operation was invalid. It also returns None to signify that the operation was not successful.

5. Docstring Added: A docstring """Performs a basic arithmetic operation based on the provided string.""" is included to explain the function's purpose.

6. Example Usage: The example usage demonstrates calling the function with different operations, including the division by zero case and an invalid operation, to show how the error handling works.

In summary, the improved version is more readable, maintainable, and robust due to better naming, consistent formatting, and proper error handling.

\* \* \* \* \*