

AI ASSISTED CODING

LAB ASSIGNMENT – 10.4

HALL TICKET NO : 2403A52403

BATCH NO : 14

Task – 01:

```
# buggy_code_task1.py
```

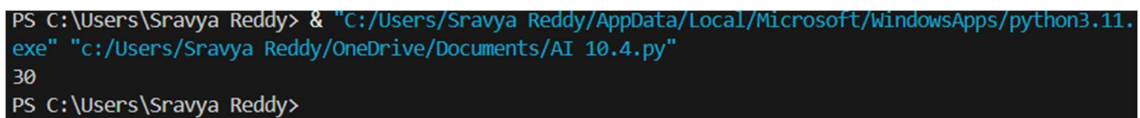
```
def add_numbers(a, b): # Added colon after function definition
```

```
    result = a + b
```

```
    return result # Corrected variable name from 'reslt' to 'result'
```

```
print(add_numbers(10, 20)) # Added comma between arguments
```

Output :



```
PS C:\Users\Sravya Reddy> & "C:/Users/Sravya Reddy/AppData/Local/Microsoft/windowsApps/python3.11.exe" "c:/Users/Sravya Reddy/OneDrive/Documents/AI 10.4.py"
30
PS C:\Users\Sravya Reddy>
```

Explanation :

Certainly! The code in cell ef2abfb5 defines a simple Python function called `add_numbers` that takes two arguments, `a` and `b`. It calculates their sum and stores it in a variable named `result`. Finally, it returns the value of `result`. The line `print(add_numbers(10, 20))` calls this function with the arguments 10 and 20, and prints the returned value, which is their sum (30).

Task – 02

```
def find_duplicates(nums):
```

```
    seen = set()
```

```
    duplicates = set()
```

```
    for num in nums:
```

```
        if num in seen:
```

```
            duplicates.add(num)
```

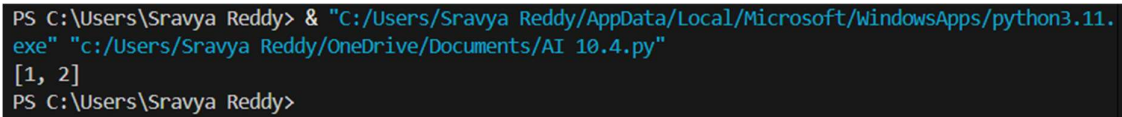
```
        else:
```

```
        seen.add(num)

    return list(duplicates)

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

Output :



```
PS C:\Users\Sravva Reddy> & "C:/Users/Sravva Reddy/AppData/Local/Microsoft/windowsApps/python3.11.exe" "c:/Users/Sravva Reddy/OneDrive/Documents/AI 10.4.py"
[1, 2]
PS C:\Users\Sravva Reddy>
```

Explanation :

1. **seen** → remembers all numbers we already looked at.
2. **duplicates** → keeps numbers that show up more than once.
3. For each number:
 - If we **already saw it**, put it in duplicates.
 - If not, add it to seen.
4. At the end, return all duplicates.

Task – 03

buggy_code_task3.py

```
def calculate_factorial(number):
```

```
    """Calculates the factorial of a non-negative integer.
```

Args:

number: The non-negative integer for which to calculate the factorial.

Returns:

The factorial of the given number.

```

"""

factorial_result = 1

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

    factorial_result = factorial_result * i

return factorial_result

print(calculate_factorial(5))

```

Output :

```

PS C:\Users\Sravva Reddy> & "C:/Users/Sravva Reddy/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Sravva Reddy/OneDrive/Documents/AI 10.4.py"
120
PS C:\Users\Sravva Reddy>

```

Explanation :

Certainly! The code in cell d9888b23 defines a function called `calculate_factorial` that computes the factorial of a non-negative integer.

Here's a breakdown:

- `def calculate_factorial(number):` defines the function that takes one argument, `number`.
- The docstring explains what the function does, its arguments, and what it returns.
- `factorial_result = 1` initializes a variable `factorial_result` to 1. This is the base case for the factorial calculation (the factorial of 0 is 1).
- `for i in range(1, number + 1):` starts a loop that iterates from 1 up to and including the number provided.
- `factorial_result = factorial_result * i` multiplies `factorial_result` by the current value of `i` in each iteration. This is the core of the factorial calculation.
- `return factorial_result` returns the final calculated factorial.
- `print(calculate_factorial(5))` calls the function with the argument 5 and prints the returned value, which is the factorial of 5 (120).

0 / 2000

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

Task – 04:

```
# enhanced_code_task4.py
```

```
import sqlite3
```

```
def get_user_data_enhanced(user_id):
```

```
    """
```

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

Args:

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

    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:

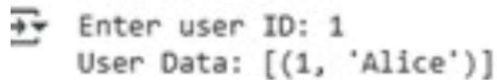
    print("User Data:", user_data)

```

else:

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

Output :

A terminal window with a dark background. The first line shows a prompt character followed by the text "Enter user ID: 1". The second line shows the output "User Data: [(1, 'Alice')]" in a light green color.

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

Explanation :

Certainly! The code in cell t08l_xt6MS4h defines a Python function `get_user_data_enhanced` that retrieves user data from an SQLite database named "users.db". This enhanced version includes security and error handling features.

Here's a breakdown:

- `import sqlite3`: Imports the necessary library for working with SQLite databases.
- `def get_user_data_enhanced(user_id)::` Defines the function that takes `user_id` as input.
- `conn = None`: Initializes the database connection variable to `None`.
- The `try...except...finally` block is used for robust error handling and resource management.
- **Input Validation**: `if not str(user_id).isdigit():` checks if the input `user_id` consists only of digits. If not, it prints an error and returns `None`. This prevents non-numeric input from causing errors or security vulnerabilities.
- `conn = sqlite3.connect("users.db")`: Establishes a connection to the database.
- `cursor = conn.cursor()`: Creates a cursor object to execute SQL commands.
- **Parameterized Query**: `query = "SELECT * FROM users WHERE id = ?;"` and `cursor.execute(query, (user_id,))`: This is a crucial security measure. Instead of directly embedding the user input into the SQL query string (which would be vulnerable to SQL injection), a placeholder `?` is used for the user ID, and the actual value is passed as a tuple to the `execute` method. The database driver handles the proper escaping of the input, preventing malicious code from being executed.
- `result = cursor.fetchall()`: Fetches all rows that match the query.
- `return result`: Returns the retrieved user data.
- `except sqlite3.Error as e::` Catches specific SQLite database errors and prints an error message.

- except Exception as e:: Catches any other unexpected errors.
- finally:: This block always executes, whether an exception occurred or not.
- if conn: conn.close(): Closes the database connection if it was successfully opened.
- The remaining code demonstrates how to create a dummy database and table for testing and how to call the get_user_data_enhanced function with user input.

In essence, this code is a safer and more robust way to retrieve data from a database based on user input compared to the original, less secure version.

0 / 2000

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

Task – 05:

improved_code_task5.py

```
def perform_operation(num1, num2, operation):
```

```
    """Performs a basic arithmetic operation on two numbers.
```

Args:

num1: The first number.

num2: The second number.

operation: The operation to perform ("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

elif operation == "div":

    if num2 == 0:

        print("Error: Division by zero is not allowed.")

        return None # Or raise a ValueError: raise ValueError("Division by zero")

    return num1 / num2

else:

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

    return None # Or raise a ValueError: raise ValueError(f"Invalid operation
'{operation}'")

```

Example usage:

```

print(perform_operation(10, 5, "add"))
print(perform_operation(10, 0, "div"))
print(perform_operation(10, 5, "mul"))
print(perform_operation(10, 5, "sub"))
print(perform_operation(10, 5, "unknown"))

```

Output :

```

PS C:\Users\Sravya Reddy> & "C:/Users/Sravya Reddy/AppData/Local/Microsoft/windowsApps/python3.11.
exe" "c:/Users/Sravya Reddy/OneDrive/Documents/AI 10.4.py"
15
Error: Division by zero is not allowed.
None
50
5
Error: Invalid operation 'unknown'.

```

Explanation :

Certainly! The code in cell 035a8e55 defines an improved Python function called `perform_operation` that takes two numbers (`num1`, `num2`) and an operation string as input, and performs the specified arithmetic operation.

Here's a breakdown:

- `def perform_operation(num1, num2, operation):`: Defines the function with three arguments.
- The docstring explains the function's purpose, arguments, and return value.
- The code uses if-elif-else statements to check the value of the operation string.
- `if operation == "add": return num1 + num2`: If the operation is "add", it returns the sum of num1 and num2.
- `elif operation == "sub": return num1 - num2`: If the operation is "sub", it returns the difference.
- `elif operation == "mul": return num1 * num2`: If the operation is "mul", it returns the product.
- `elif operation == "div":`: If the operation is "div", it includes an additional check:
 - `if num2 == 0:` If the second number (num2) is 0, it prints an error message for division by zero and returns None. This prevents a ZeroDivisionError.
 - `return num1 / num2`: If num2 is not 0, it returns the result of the division.
- `else: print(f"Error: Invalid operation '{operation}'); return None`: If the operation string does not match any of the valid operations, it prints an error message indicating an invalid operation and returns None.
- The example usage lines demonstrate calling the function with different numbers and operations, including a division by zero case and an invalid operation case, showing how the error handling works.

This improved version is more robust as it handles division by zero and invalid operations gracefully, and it uses more descriptive names and includes a docstring for better readability compared to the original code.