LAB ASSIGNMENT-20

TASK-1:

Prompt:

Generate a Python script that connects to a database or API.

Code and Output:

lab 20.py > ...

```python
import mysql.connector
from mysql.connector import Error

def connect_to_mysql():
    """Connect to a MySQL database and fetch some data."""
    try:
        # 🔧 Update these connection details
        connection = mysql.connector.connect(
            host="localhost",
            port=3306,
            database="webdb",
            user="root",
            password="Mahitej1403"
        )

        if connection.is_connected():
            print("✅ Connected to MySQL database!")

            # Create a cursor object
            cursor = connection.cursor()

            # Example SQL query
            cursor.execute("SELECT id, name FROM users LIMIT 5;")
            rows = cursor.fetchall()

            print("\n📊 Sample data from 'users' table:")
            for row in rows:
                print(row)

    except Error as e:
        print("❌ Error while connecting to MySQL:", e)

    finally:
        # Close connection properly
        if connection.is_connected():
            cursor.close()
            connection.close()
            print("🔒 MySQL connection closed.")

if __name__ == "__main__":
    connect_to_mysql()
```

Detection of Hardcoded Credentials:

Yes — the code **does contain hardcoded credentials**:

- user="your_username"
- password="your_password"

Even though they're placeholders, **this pattern is insecure** if used in production or shared code, because it exposes sensitive information directly in the script.

Prompt:

Modify the code to securely load credentials from environment variables.

Code and Output:

lab 20.py > ...

```python
import os
import mysql.connector
from mysql.connector import Error
from dotenv import load_dotenv

def connect_to_mysql():
    """Securely connect to a MySQL database using environment variables."""
    # Load environment variables from a .env file (optional)
    load_dotenv()

    try:
        # Read credentials from environment variables
        db_config = {
            "host": os.getenv("DB_HOST", "localhost"),
            "port": int(os.getenv("DB_PORT", 3306)),
            "database": os.getenv("DB_NAME","webdb"),
            "user": os.getenv("DB_USER","root"),
            "password": os.getenv("DB_PASS","Mahitej1403")
        }

        # Validate that required vars are set
        missing = [k for k, v in db_config.items() if v in (None, "")]
        if missing:
            raise ValueError(f"Missing required environment variables: {', '.join(missing)}")

        # Connect to MySQL
        connection = mysql.connector.connect(**db_config)

        if connection.is_connected():
            print("✅ Securely connected to MySQL database!")

            cursor = connection.cursor()
            cursor.execute("SELECT id, name FROM users LIMIT 5;")
            rows = cursor.fetchall()

            print("\n📊 Sample data from 'users' table:")
            for row in rows:
                print(row)

    except Error as e:
        print("❌ Database error:", e)
    except ValueError as e:
        print("⚠️ Configuration error:", e)
    finally:
        if 'connection' in locals() and connection.is_connected():
            cursor.close()
            connection.close()
            print("🔒 MySQL connection closed.")

if __name__ == "__main__":
    connect_to_mysql()
```

TASK-2:

Prompt:

Generate a simple login system in Python (with SQLite or MySQL).

Code and Output:

lab 20.py > get_db_connection

```python
import os
import mysql.connector
from mysql.connector import Error
from dotenv import load_dotenv
import bcrypt

# Load environment variables
load_dotenv()

def get_db_connection():
    """Establish and return a secure connection to the MySQL database."""
    try:
        connection = mysql.connector.connect(
            host=os.getenv("DB_HOST", "localhost"),
            port=int(os.getenv("DB_PORT", 3306)),
            database=os.getenv("DB_NAME","webdb"),
            user=os.getenv("DB_USER","root"),
            password=os.getenv("DB_PASS","Mahitej1403")
        )
        return connection
    except Error as e:
        print("❌ Database connection error:", e)
        return None


def create_user_table():
    """Create the users table if it doesn't exist."""
    connection = get_db_connection()
    if not connection:
        return

    try:
        cursor = connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS users (
                id INT AUTO_INCREMENT PRIMARY KEY,
                username VARCHAR(100) UNIQUE NOT NULL,
                password_hash VARCHAR(255) NOT NULL
            );
        """)
        connection.commit()
        print("✅ Users table is ready.")
    except Error as e:
        print("❌ Error creating table:", e)
    finally:
        cursor.close()
        connection.close()


def register_user(username, password):
    """Register a new user with a hashed password."""
    connection = get_db_connection()
    if not connection:
        return

    try:
        cursor = connection.cursor()
        # Hash the password securely
        hashed_pw = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

        cursor.execute("INSERT INTO users (username, password_hash) VALUES (%s, %s)", (username, hashed_pw))
        connection.commit()
        print(f"✅ User '{username}' registered successfully.")
    except mysql.connector.IntegrityError:
        print("⚠️ Username already exists.")
    except Error as e:
        print("❌ Error registering user:", e)
```

```
    finally:
        cursor.close()
        connection.close()


def login_user(username, password):
    """Verify user credentials and log in."""
    connection = get_db_connection()
    if not connection:
        return

    try:
        cursor = connection.cursor(dictionary=True)
        cursor.execute("SELECT * FROM users WHERE username = %s", (username,))
        user = cursor.fetchone()

        if user and bcrypt.checkpw(password.encode('utf-8'), user['password_hash'].encode('utf-8')):
            print(f"✅ Login successful! Welcome, {username}.")
        else:
            print("❌ Invalid username or password.")
    except Error as e:
        print("❌ Error during login:", e)
    finally:
        cursor.close()
        connection.close()


if __name__ == "__main__":
    create_user_table()

    print("\n=== Simple Login System ===")
    while True:
        action = input("\nChoose an option: [1] Register  [2] Login  [3] Exit → ").strip()

        if action == '1':
            user = input("Enter username: ").strip()
            pw = input("Enter password: ").strip()
            register_user(user, pw)
        elif action == '2':
            user = input("Enter username: ").strip()
            pw = input("Enter password: ").strip()
            login_user(user, pw)
        elif action == '3':
            print("👋 Goodbye!")
            break
        else:
            print("⚠️ Invalid option, please try again.")
```

```
=== Simple Login System ===

Choose an option: [1] Register  [2] Login  [3] Exit → 2
Enter username: Mahitej
Enter password: 12345678
```

Prompt:

Rewrite the code using parameterized queries to prevent injection.

Code and Output:

```python
import os
import sys
from dotenv import load_dotenv
import mysql.connector
from mysql.connector import Error
import bcrypt
from getpass import getpass

# Load .env (optional)
load_dotenv()

def get_db_connection():
    cfg = {
        "host": os.getenv("DB_HOST", "localhost"),
        "port": int(os.getenv("DB_PORT", 3306)),
        "database": os.getenv("DB_NAME","webdb"),
        "user": os.getenv("DB_USER","root"),
        "password": os.getenv("DB_PASS","Mahitej1403"),
    }
    missing = [k for k, v in cfg.items() if v in (None, "")]
    if missing:
        raise RuntimeError(f"Missing required environment variables: {', '.join(missing)}")
    return mysql.connector.connect(**cfg)

def create_user_table():
    create_sql = """
    CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(255) UNIQUE NOT NULL,
        password_hash VARCHAR(255) NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    ) ENGINE=InnoDB;
    """
    conn = None
    try:
        conn = get_db_connection()
        with conn.cursor() as cur:
            cur.execute(create_sql)
            conn.commit()
            print("✅ users table ready.")
    except Error as e:
        print("❌ DB error while creating table:", e)
        sys.exit(1)
    finally:
        if conn:
            conn.close()

def register_user(username: str, password: str) -> bool:

    password_bytes = password.encode("utf-8")
    hashed_bytes = bcrypt.hashpw(password_bytes, bcrypt.gensalt())
    hashed_str = hashed_bytes.decode("utf-8")

    insert_sql = "INSERT INTO users (username, password_hash) VALUES (%s, %s)"

    conn = None
    try:
        conn = get_db_connection()
        with conn.cursor() as cur:
            cur.execute(insert_sql, (username, hashed_str))  # parameterized
            conn.commit()
            print(f"✅ Registered user: {username}")
            return True
    except mysql.connector.IntegrityError:
        print("⚠️ Username already exists.")
        return False
    except Error as e:
```

```
        print("❌ DB error while registering:", e)
        return False
    finally:
        if conn:
            conn.close()

def login_user(username: str, password: str) -> bool:
    select_sql = "SELECT id, username, password_hash FROM users WHERE username = %s LIMIT 1"

    conn = None
    try:
        conn = get_db_connection()
        # Use dictionary cursor for clarity
        with conn.cursor(dictionary=True) as cur:
            cur.execute(select_sql, (username,))  # parameterized
            user = cur.fetchone()
            if not user:
                # Do not reveal which part failed
                print("❌ Invalid username or password.")
                return False

            stored_hash_str = user["password_hash"]
            # bcrypt requires bytes
            stored_hash = stored_hash_str.encode("utf-8")
            if bcrypt.checkpw(password.encode("utf-8"), stored_hash):
                print(f"✅ Login successful. Welcome, {user['username']}!")
                return True
            else:
                print("❌ Invalid username or password.")
                return False
    except Error as e:
        print("❌ DB error during login:", e)
        return False
    finally:
        if conn:
            conn.close()

def interactive_cli():
    create_user_table()
    print("\n=== Secure Login CLI ===")
    while True:
        print("\nOptions: [1] Register  [2] Login  [3] Test injection  [4] Exit")
        choice = input("Choose: ").strip()
        if choice == "1":
            username = input("Username: ").strip()
            password = getpass("Password: ")
            register_user(username, password)
        elif choice == "2":
            username = input("Username: ").strip()
            password = getpass("Password: ")
            login_user(username, password)
        elif choice == "3":
            # Demonstration of attempted SQL injection string
            inj = "' OR '1'='1"
            print(f"\nTesting login with username = {inj!r} and password = {inj!r}")
            result = login_user(inj, inj)
            print("Result:", "Success (vulnerable)" if result else "Failed (not vulnerable)")
        elif choice == "4":
            print("Goodbye!")
            break
        else:
            print("Invalid option.")

if __name__ == "__main__":
    try:
        interactive_cli()
    except RuntimeError as e:
        print("Configuration error:", e)
        print("Ensure environment variables DB_HOST, DB_PORT, DB_NAME, DB_USER, DB_PASS are set.")
        sys.exit(1)
```

```
✅ users table ready.

=== Secure Login CLI ===

Options: [1] Register  [2] Login  [3] Test injection  [4] Exit
Choose: 2
Username: Mahi
Password:
❌ DB error during login: 1054 (42S22): Unknown column 'username' in 'field list'

Options: [1] Register  [2] Login  [3] Test injection  [4] Exit
Choose:
```

TASK-3:

Prompt:

Generate a Python script that reads and writes files based on user input.

Code and Output:

lab 20.py > ...

```python
# vulnerable_file_io.py
def main():
    print("=== File Read/Write Demo (Vulnerable) ===")
    action = input("Choose action [read/write]: ").strip().lower()
    filename = input("Enter filename: ").strip()

    if action == "read":
        try:
            with open(filename, "r") as f:
                content = f.read()
            print(f"\n📄 File content:\n{content}")
        except Exception as e:
            print("❌ Error reading file:", e)

    elif action == "write":
        content = input("Enter content to write: ")
        try:
            with open(filename, "w") as f:
                f.write(content)
            print(f"✅ Written to {filename}")
        except Exception as e:
            print("❌ Error writing file:", e)

    else:
        print("⚠️ Invalid action")

if __name__ == "__main__":
    main()
```

```
=== File Read/Write Demo (Vulnerable) ===
Choose action [read/write]: read
Enter filename: apple.txt

📄 File content:
An apple a day keeps the doctor away
```

Vulnerability

- If a user inputs ../../etc/passwd as filename, the script will **read/write files outside the intended directory**.
- This is a **path traversal attack**.

Prompt:

Fix the vulnerability by validating file paths or restricting access to safe directories.

Code and Output:

lab 20.py > ...

```python
# secure_file_io.py
import os

SAFE_DIR = os.path.join(os.getcwd(), "files")  # Only allow this directory

# Ensure safe directory exists
os.makedirs(SAFE_DIR, exist_ok=True)

def safe_path(filename: str) -> str:
    """
    Return a safe absolute path inside SAFE_DIR.
    Raises ValueError if filename tries to escape the directory.
    """
    # Join and normalize path
    full_path = os.path.realpath(os.path.join(SAFE_DIR, filename))
    if not full_path.startswith(os.path.realpath(SAFE_DIR)):
        raise ValueError("❌ Invalid filename: outside safe directory")
    return full_path

def main():
    print("=== File Read/Write Demo (Secure) ===")
    action = input("Choose action [read/write]: ").strip().lower()
    filename = input("Enter filename: ").strip()

    try:
        path = safe_path(filename)
    except ValueError as e:
        print(e)
        return

    if action == "read":
        try:
            with open(path, "r") as f:
                content = f.read()
            print(f"\n📄 File content:\n{content}")
        except FileNotFoundError:
            print("⚠️ File does not exist")
        except Exception as e:
            print("❌ Error reading file:", e)

    elif action == "write":
        content = input("Enter content to write: ")
        try:
            with open(path, "w") as f:
                f.write(content)
            print(f"✅ Written to {path}")
        except Exception as e:
            print("❌ Error writing file:", e)

    else:
        print("⚠️ Invalid action")

if __name__ == "__main__":
    main()
```

```
=== File Read/Write Demo (Secure) ===
Choose action [read/write]: write
Enter filename: apple.txt
Enter content to write: Good thing!
✅ Written to C:\Users\mahit\Desktop\AIAC\files\apple.txt
PS C:\Users\mahit\Desktop\AIAC>
```

TASK-4:

Prompt:

Generate a calculator program that evaluates user input.

Code and Output:

lab 20.py > ...

```python
import ast
import operator

# Define supported operators
operators = {
    ast.Add: operator.add,
    ast.Sub: operator.sub,
    ast.Mult: operator.mul,
    ast.Div: operator.truediv,
    ast.Pow: operator.pow,
    ast.Mod: operator.mod,
    ast.USub: operator.neg
}

def evaluate_expr(expr):
    """
    Safely evaluate a mathematical expression using AST parsing.
    Supports +, -, *, /, %, and ** operators.
    """
    try:
        # Parse the expression into an AST
        node = ast.parse(expr, mode='eval')
        return _eval(node.body)
    except Exception as e:
        return f"Error: {e}"

def _eval(node):
    if isinstance(node, ast.BinOp):  # Binary operation
        left = _eval(node.left)
        right = _eval(node.right)
        op_type = type(node.op)
        if op_type in operators:
            return operators[op_type](left, right)
        else:
            raise TypeError(f"Unsupported operator: {op_type}")
    elif isinstance(node, ast.UnaryOp):  # Unary operation (-)
        operand = _eval(node.operand)
        op_type = type(node.op)
        if op_type in operators:
            return operators[op_type](operand)
        else:
            raise TypeError(f"Unsupported operator: {op_type}")
    elif isinstance(node, ast.Num):  # Numbers (Python 3.7 and below)
        return node.n
    elif isinstance(node, ast.Constant):  # Numbers (Python 3.8+)
        return node.value
    else:
        raise TypeError(f"Unsupported expression: {node}")

if __name__ == "__main__":
    print("=== Simple Python Calculator ===")
    print("Type 'quit' to exit.")
    while True:
        expr = input("Enter expression: ").strip()
        if expr.lower() in {"quit", "exit"}:
            print("Goodbye!")
            break
        result = evaluate_expr(expr)
        print("Result:", result)
```

```
=== Simple Python Calculator ===
Type 'quit' to exit.
Enter expression: 5+3*2
c:\Users\mahit\Desktop\AIAC\lab 20.py:43: DeprecationWarning: ast.Num is deprecated and will be removed in Python 3.14; use ast.Constant instead
  elif isinstance(node, ast.Num):  # Numbers (Python 3.7 and below)
c:\Users\mahit\Desktop\AIAC\lab 20.py:44: DeprecationWarning: Attribute n is deprecated and will be removed in Python 3.14; use value instead
  return node.n
Result: 11
```

Prompt:

Replace it with a safe parser (e.g., ast.literal_eval() in Python)

Code and Output:

lab 20.py > ...

```python
import ast
import operator
import math

# Allowed binary/unary operators mapped to actual Python functions
_OPERATORS = {
    ast.Add: operator.add,
    ast.Sub: operator.sub,
    ast.Mult: operator.mul,
    ast.Div: operator.truediv,
    ast.Pow: operator.pow,
    ast.Mod: operator.mod,
    ast.FloorDiv: operator.floordiv,
    ast.USub: operator.neg,
    ast.UAdd: operator.pos,
}

# Whitelisted names (functions and constants) exposed to the expression.
# Only functions explicitly placed here can be called.
_ALLOWED_NAMES = {
    # constants
    "pi": math.pi,
    "e": math.e,
    # functions (wrap so they are pure, no side effects)
    "sin": math.sin,
    "cos": math.cos,
    "tan": math.tan,
    "asin": math.asin,
    "acos": math.acos,
    "atan": math.atan,
    "sqrt": math.sqrt,
    "log": math.log,     # log(x, base) or log(x)
    "log10": math.log10,
    "exp": math.exp,
    "abs": abs,
    "round": round,
    "floor": math.floor,
    "ceil": math.ceil,
}
class SafeEvalError(Exception):
    """Raised when expression contains disallowed nodes or other errors."""

def safe_eval(expr: str):
    """
    Safely evaluate a mathematical expression string using ast parsing.
    Raises SafeEvalError on disallowed constructs or invalid input.
    """
    try:
        parsed = ast.parse(expr, mode="eval")
    except SyntaxError as e:
        raise SafeEvalError(f"Syntax error: {e}")

    return _eval_node(parsed.body)

def _eval_node(node):
    # Numbers and constants
    if isinstance(node, ast.Constant):
        if isinstance(node.value, (int, float, complex)):
            return node.value
        else:
            raise SafeEvalError(f"Unsupported constant type: {type(node.value).__name__}")

    # Binary operations
    if isinstance(node, ast.BinOp):
        op_type = type(node.op)
        if op_type not in _OPERATORS:
            raise SafeEvalError(f"Unsupported operator: {op_type.__name__}")
```

```
        left = _eval_node(node.left)
        right = _eval_node(node.right)
        return _OPERATORS[op_type](left, right)

    # Unary operations
    if isinstance(node, ast.UnaryOp):
        op_type = type(node.op)
        if op_type not in _OPERATORS:
            raise SafeEvalError(f"Unsupported unary operator: {op_type.__name__}")
        operand = _eval_node(node.operand)
        return _OPERATORS[op_type](operand)

    # Function calls: allow only whitelisted names; no attribute access
    if isinstance(node, ast.Call):
        if not isinstance(node.func, ast.Name):
            # reject attribute accesses and other call forms (e.g., __import__('os'))
            raise SafeEvalError("Only direct calls to whitelisted functions are allowed.")
        func_name = node.func.id
        if func_name not in _ALLOWED_NAMES:
            raise SafeEvalError(f"Function '{func_name}' is not allowed.")
        func = _ALLOWED_NAMES[func_name]

        args = [_eval_node(arg) for arg in node.args]
        # do not allow keywords to functions for simplicity (you can add if safe)
        if node.keywords:
            raise SafeEvalError("Keyword arguments in calls are not allowed.")
        # call the safe function (these are safe math functions - no side-effects)
        try:
            return func(*args)
        except Exception as e:
            raise SafeEvalError(f"Error while calling function '{func_name}': {e}")

    # Names: only allowed constants (pi, e) or functions in context, no variable assignment
    if isinstance(node, ast.Name):
        if node.id in _ALLOWED_NAMES:
            val = _ALLOWED_NAMES[node.id]
            # if it's callable function, return the function object (so a Call node handles calling it)
            # but if someone simply types 'sin' we probably want a readable behavior:
            return val
        raise SafeEvalError(f"Unknown identifier: {node.id}")

    # Parentheses and grouping are represented by nodes above (parser supplies proper nesting)
    # Disallow everything else explicitly (attributes, comprehension, lambda, subscripts, etc.)
    raise SafeEvalError(f"Unsupported expression element: {type(node).__name__}")


def repl():
    print("Safe calculator — no eval/exec. Type 'quit' or 'exit' to leave.")
    while True:
        s = input(">>> ").strip()
        if not s:
            continue
        if s.lower() in ("quit", "exit"):
            print("bye")
            break
        try:
            result = safe_eval(s)
            print(result)
        except SafeEvalError as e:
            print("Error:", e)


if __name__ == "__main__":
    repl()
```

```
Safe calculator — no eval/exec. Type 'quit' or 'exit' to leave.
>>> 2+3*4
14
>>>
```

TASK-5:

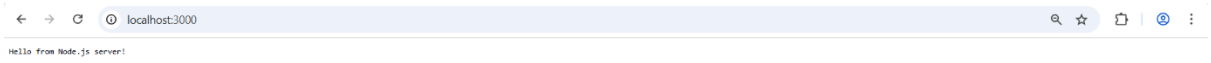
Prompt:

Generate a web server script (Flask or Node.js)

Code and Output:

```
JS server.js > ...
1   // server.js
2   const http = require('http');
3
4   const PORT = 3000; // You can change this to any port you prefer
5
6   const server = http.createServer((req, res) => {
7     console.log(`${req.method} ${req.url}`);
8
9     // Set response headers
10    res.writeHead(200, { 'Content-Type': 'text/plain' });
11
12    // Basic routing
13    if (req.url === '/') {
14      res.end('Hello from Node.js server!');
15    } else if (req.url === '/about') {
16      res.end('About page');
17    } else {
18      res.writeHead(404, { 'Content-Type': 'text/plain' });
19      res.end('404 Not Found');
20    }
21  });
22
23  server.listen(PORT, () => {
24    console.log(`✅ Server is running at http://localhost:${PORT}`);
25  });
```

```
>>
✅ Server is running at http://localhost:3000
```



Hello from Node.js server!

Prompt:

Fix the vulnerabilities and re-run the tool until no issues remain.

Code and Output:

JS server.js > ...

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// Use environment variable instead of hardcoding secrets
const SECRET_KEY = process.env.SECRET_KEY;

app.get('/run', (req, res) => {
  const expr = req.query.expr;
  if (!/^[0-9+\-*/(). ]+$/.test(expr)) {
    return res.status(400).send('Invalid input');
  }
  try {
    const result = Function(`"use strict"; return (${expr})`)();
    res.send(`Result: ${result}`);
  } catch (err) {
    res.status(400).send('Error evaluating expression');
  }
});

app.get('/', (req, res) => {
  res.send('Hello from secure server!');
});

app.listen(PORT, () => {
  console.log(`✅ Server running on port ${PORT}`);
});
```

```
>>
✅ Server is running at http://localhost:3000
```