AI-ASSISTED CODING

NAME : ANJALI. JEELUKA

HT.NO : 2403A510F0

BATCH :05

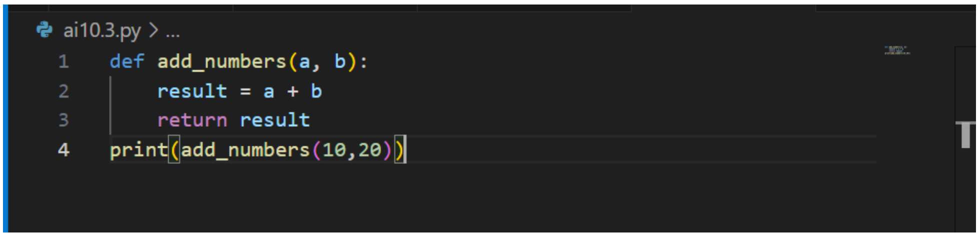
DEPT:CSE

TASK-1:

- **PROMPT :**
  Identify and fix syntax, indentation, and variable errors in a given Python script. Provide the corrected code and clearly explain what errors were fixed (such as missing colons, indentation issues, variable name typos, or incorrect function calls).

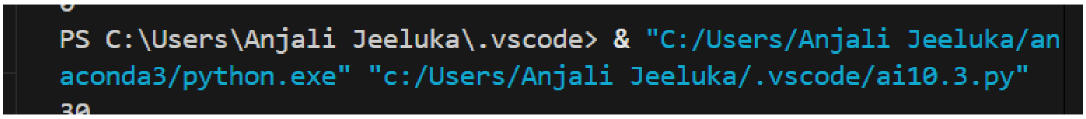
```
  #buggy_code
def add_numbers(a, b)
result = a + b
return reslt
print(add_numbers(10 20))
```

- **CODE :**



```python
def add_numbers(a, b):
    result = a + b
    return result
print(add_numbers(10,20))
```

OUTPUT :



```
PS C:\Users\Anjali Jeeluka\.vscode> & "C:/Users/Anjali Jeeluka/an
aconda3/python.exe" "c:/Users/Anjali Jeeluka/.vscode/ai10.3.py"
30
```

- **EXPLANATION:**

- Added a colon (:) at the end of the function definition line.
- Indented the function body correctly.
- Fixed the typo reslt to result in the return statement.
- Added a comma between 10 and 20 in the function call.

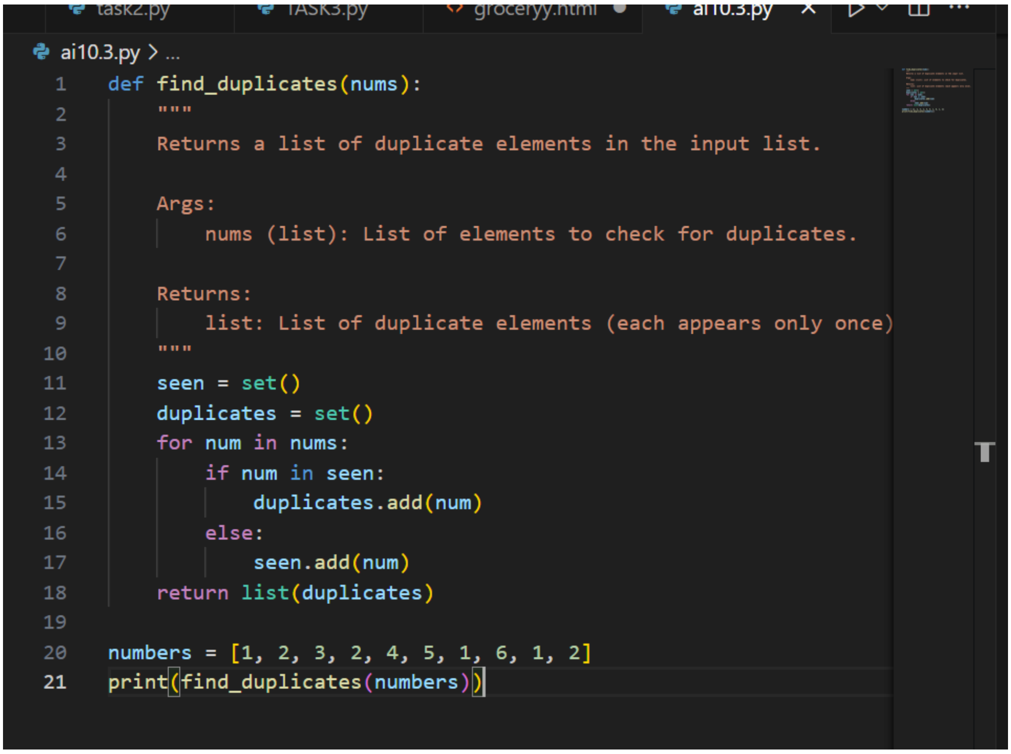
TASK-2 :

- **PROMPT :**

  Can you review a Python function that finds duplicates in a list, identify the logical and performance issues (such as nested loops or inefficient membership checks), and then rewrite the function to optimize its performance using a more efficient approach like sets, while keeping the output correct? Also, explain the optimization made .

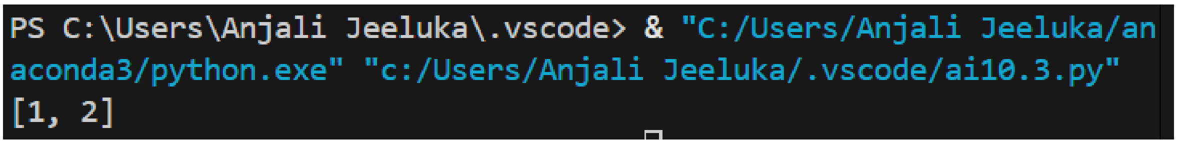
```
#buggy_code
def find_duplicates(nums):
duplicates = []
for i in range(len(nums)):
for j in range(len(nums)):
if i != j and nums[i] == nums[j] and nums[i] not in duplicates:
duplicates.append(nums[i])
return duplicates
numbers = [1,2,3,2,4,5,1,6,1,2]
print(find_duplicates(numbers))
```

➢ CODE :



```python
def find_duplicates(nums):
    """
    Returns a list of duplicate elements in the input list.

    Args:
        nums (list): List of elements to check for duplicates.

    Returns:
        list: List of duplicate elements (each appears only once)
    """
    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\Anjali Jeeluka\.vscode> & "C:/Users/Anjali Jeeluka/an
aconda3/python.exe" "c:/Users/Anjali Jeeluka/.vscode/ai10.3.py"
[1, 2]
```

➢ EXPLAINATION:

- **Issues in the Original Function:**
- **Inefficient Nested Loops:** The function uses two nested loops, resulting in O(n²) time complexity.
- **Redundant Membership Checks:** nums[i] not in duplicates is checked in every iteration, making it even slower.
- **Unnecessary Comparisons:** Each pair is compared twice (i, j) and (j, i), and self-comparison is avoided with i != j.

Optimization Explanation:

- Single Loop: Only one pass through the list (O(n) time complexity).
- Set Lookup: Checking membership in a set is O(1), much faster than in a list.
- Duplicates Set: Ensures each duplicate is only added once.

Result: The function is now efficient and returns the correct list of duplicates.

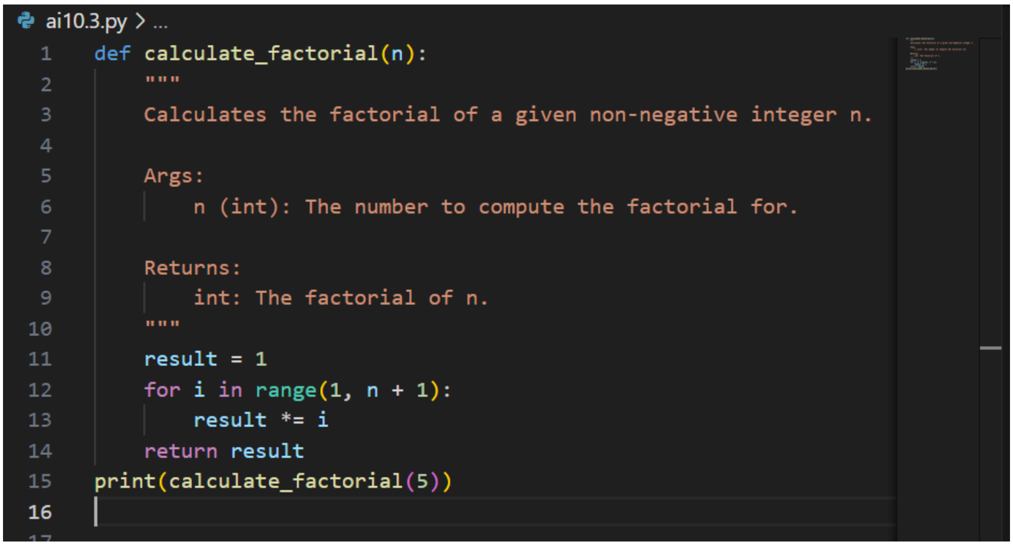
TASK-3:

- **PROMPT:**

  Can you refactor a messy Python function into clean, PEP 8–compliant, and well-structured code by renaming the function to calculate_factorial, improving variable names, adding proper indentation, formatting, and a docstring to explain the function? #buggy_code

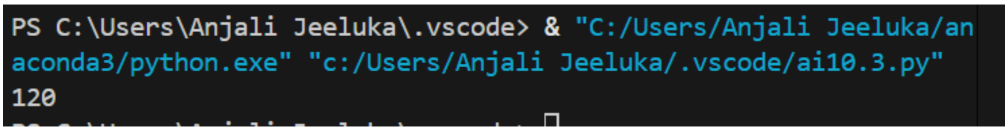
  ```
  def c(n):
  x=1
  for i in range(1,n+1):
  x=x*i
  return x
  print(c(5)
  ```

- **CODE :**



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

    Args:
        n (int): The number to compute the factorial for.

    Returns:
        int: The factorial of n.
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
print(calculate_factorial(5))
```

- OUTPUT:



```
PS C:\Users\Anjali Jeeluka\.vscode> & "C:/Users/Anjali Jeeluka/an
aconda3/python.exe" "c:/Users/Anjali Jeeluka/.vscode/ai10.3.py"
120
```

- EXPLAINATION:
- Renamed the function to calculate_factorial for clarity.
- Used descriptive variable names (result instead of x).
- Added a clear, Google-style docstring.
- Fixed indentation and formatting to comply with PEP 8.
- Ensured the print statement uses the new function name and proper parentheses.

TASK-4:

- **PROMPT:**

  How can you refactor the given Python code to make it more secure and reliable by adding safe database practices and error handling? Specifically, ensure the following:

  Replace raw SQL string formatting with parameterized queries (? placeholders) to prevent SQL injection.

  Validate user input before executing the query.

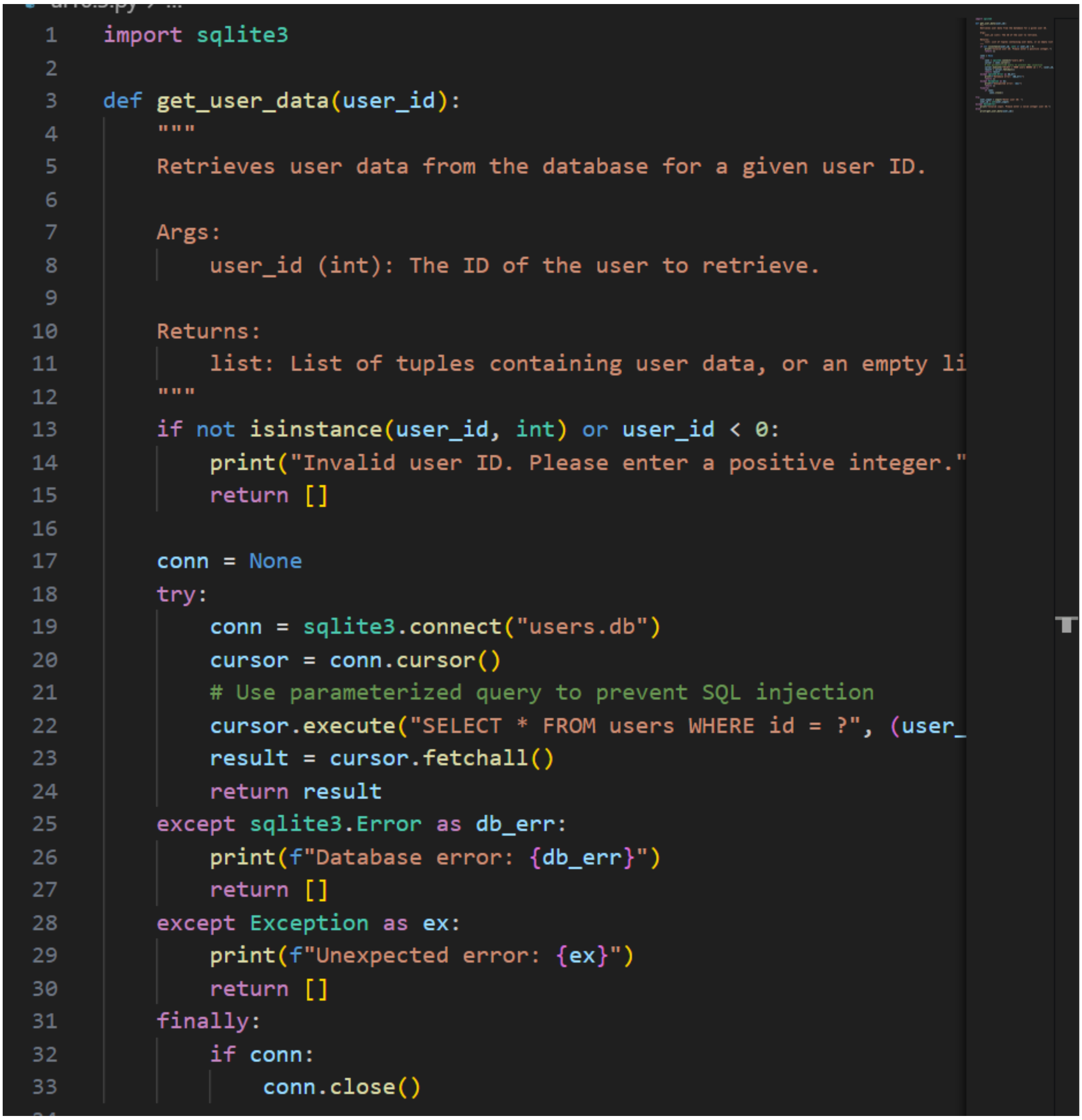
  Add try-except blocks to handle database errors, invalid inputs, and unexpected exceptions.

  Ensure the database connection is always closed safely.

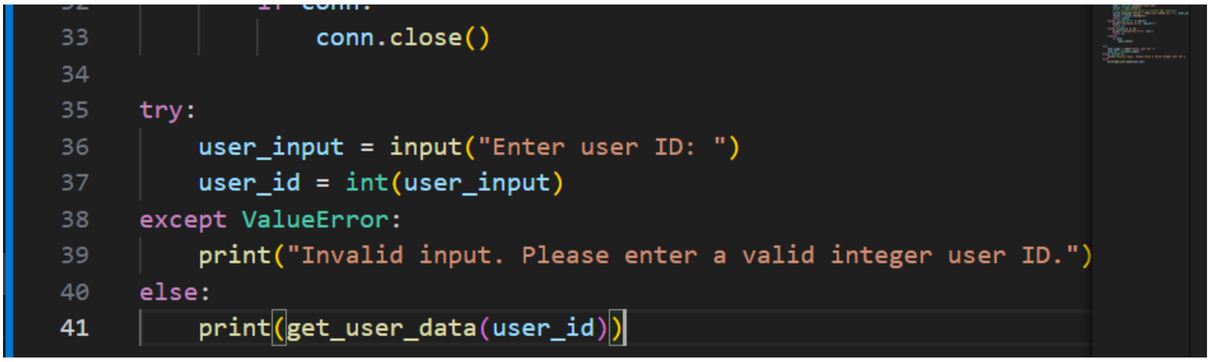
```
#buggy_code
import sqlite3
def get_user_data(user_id):
conn = sqlite3.connect("users.db")
cursor = conn.cursor()
query = f"SELECT * FROM users WHERE id = {user_id};" #
```

```
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))
```

- **CODE:**

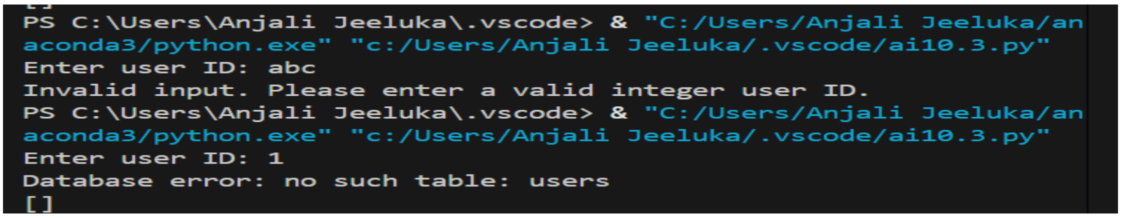


```python
import sqlite3

def get_user_data(user_id):
    """
    Retrieves user data from the database for a given user ID.

    Args:
        user_id (int): The ID of the user to retrieve.

    Returns:
        list: List of tuples containing user data, or an empty li
    """
    if not isinstance(user_id, int) or user_id < 0:
        print("Invalid user ID. Please enter a positive integer."
        return []

    conn = None
    try:
        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()
        # Use parameterized query to prevent SQL injection
        cursor.execute("SELECT * FROM users WHERE id = ?", (user_
        result = cursor.fetchall()
        return result
    except sqlite3.Error as db_err:
        print(f"Database error: {db_err}")
        return []
    except Exception as ex:
        print(f"Unexpected error: {ex}")
        return []
    finally:
        if conn:
            conn.close()
```



```python
            conn.close()

try:
    user_input = input("Enter user ID: ")
    user_id = int(user_input)
except ValueError:
    print("Invalid input. Please enter a valid integer user ID.")
else:
    print(get_user_data(user_id))
```

- **OUTPUT:**



```
PS C:\Users\Anjali Jeeluka\.vscode> & "C:/Users/Anjali Jeeluka/an
aconda3/python.exe" "c:/Users/Anjali Jeeluka/.vscode/ai10.3.py"
Enter user ID: abc
Invalid input. Please enter a valid integer user ID.
PS C:\Users\Anjali Jeeluka\.vscode> & "C:/Users/Anjali Jeeluka/an
aconda3/python.exe" "c:/Users/Anjali Jeeluka/.vscode/ai10.3.py"
Enter user ID: 1
Database error: no such table: users
[]
```

EXPLAINATION:

Improvements made:

- Used parameterized queries to prevent SQL injection.
- Validated that user_id is a positive integer before querying.
- Added try-except blocks for database and general errors.
- Ensured the database connection is always closed using a finally block.
- Provided user-friendly error messages for invalid input and exceptions.

TASK-5:

- **PROMPT:**
  How can you refactor the given Python code to make it more secure and reliable by adding safe database practices and error handling? Specifically, ensure the following:

Replace raw SQL string formatting with parameterized queries (? placeholders) to prevent SQL injection.

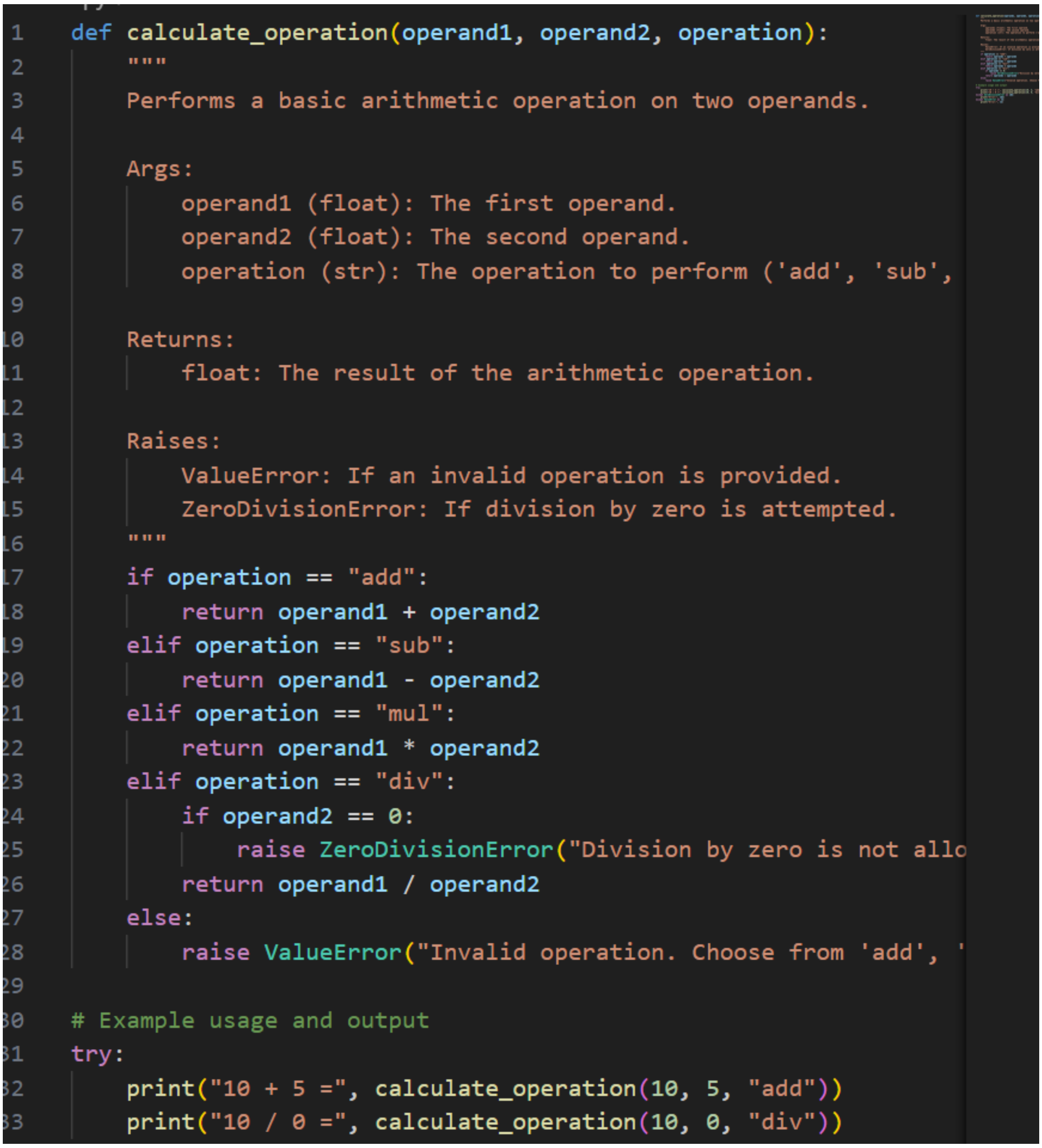
Validate user input before executing the query.

Add try-except blocks to handle database errors, invalid inputs, and unexpected exceptions.

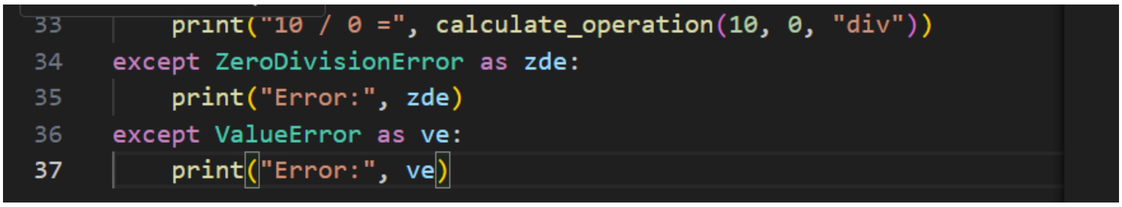
Ensure the database connection is always closed safely.

```
#buggy_code
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))
```

- **CODE :**

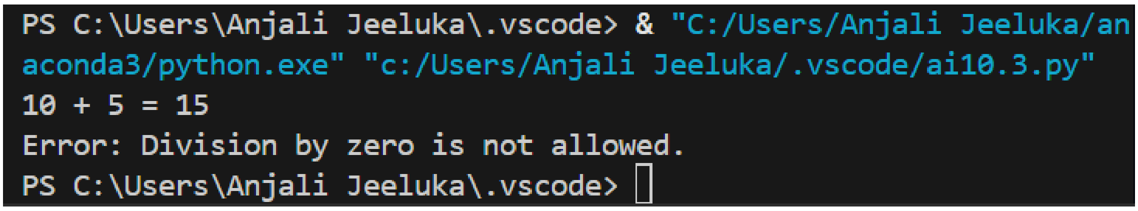


```python
def calculate_operation(operand1, operand2, operation):
    """
    Performs a basic arithmetic operation on two operands.

    Args:
        operand1 (float): The first operand.
        operand2 (float): The second operand.
        operation (str): The operation to perform ('add', 'sub',

    Returns:
        float: The result of the arithmetic operation.

    Raises:
        ValueError: If an invalid operation is provided.
        ZeroDivisionError: If division by zero is attempted.
    """
    if operation == "add":
        return operand1 + operand2
    elif operation == "sub":
        return operand1 - operand2
    elif operation == "mul":
        return operand1 * operand2
    elif operation == "div":
        if operand2 == 0:
            raise ZeroDivisionError("Division by zero is not allo
        return operand1 / operand2
    else:
        raise ValueError("Invalid operation. Choose from 'add', '

# Example usage and output
try:
    print("10 + 5 =", calculate_operation(10, 5, "add"))
    print("10 / 0 =", calculate_operation(10, 0, "div"))
```



```python
    print("10 / 0 =", calculate_operation(10, 0, "div"))
except ZeroDivisionError as zde:
    print("Error:", zde)
except ValueError as ve:
    print("Error:", ve)
```

- **OUTPUT :**



```
PS C:\Users\Anjali Jeeluka\.vscode> & "C:/Users/Anjali Jeeluka/an
aconda3/python.exe" "c:/Users/Anjali Jeeluka/.vscode/ai10.3.py"
10 + 5 = 15
Error: Division by zero is not allowed.
PS C:\Users\Anjali Jeeluka\.vscode>
```

- **EXPLAINATION :**

- Function Purpose:
  The calculate_operation function performs a basic arithmetic operation (add, sub, mul, or div) on two operands.
- Parameters:
    - operand1 (float): The first number.
    - operand2 (float): The second number.
    - operation (str): The operation to perform. Must be one of 'add', 'sub', 'mul', or 'div'.
- Return Value:
  Returns the result of the arithmetic operation as a float.
- Error Handling:
    - Raises a ZeroDivisionError if division by zero is attempted.
    - Raises a ValueError if an invalid operation string is provided.
- Usage Example:
  The code demonstrates usage by trying to
  add 10 and 5 (prints 15), and then tries to divide 10 by 0, which triggers a ZeroDivisionError and prints an error message.
- Best Practices:
    - Includes a clear, Google-style docstring.
    - Uses descriptive variable and function names.
    - Handles errors gracefully with try-except blocks.
    - Follows PEP 8 formatting and indentation guidelines.