AI ASSISTED CODING

NAME: E.HAMSITHA

ENROLL NUMBER: 2403A52361

BATCH NUMBER :13

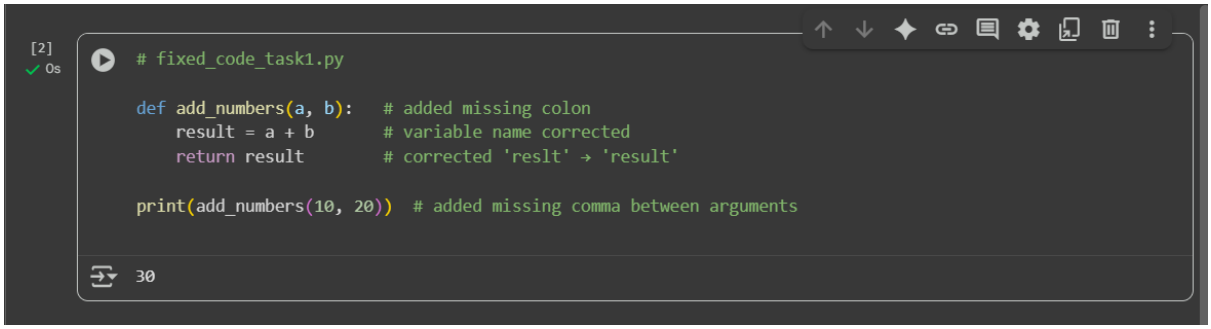
Lab assignment-10.4

Task 1: Syntax and Error Detection

Task: Identify and fix syntax, indentation, and variable errors in the given script.

```
# buggy_code_task1.py
def add_numbers(a, b)
    result = a + b
    return reslt
print(add_numbers(10 20))
```

code:



Fixes made:

- Added : after def add_numbers(a, b)
- Corrected variable typo reslt → result
- Added missing comma in print(add_numbers(10, 20))

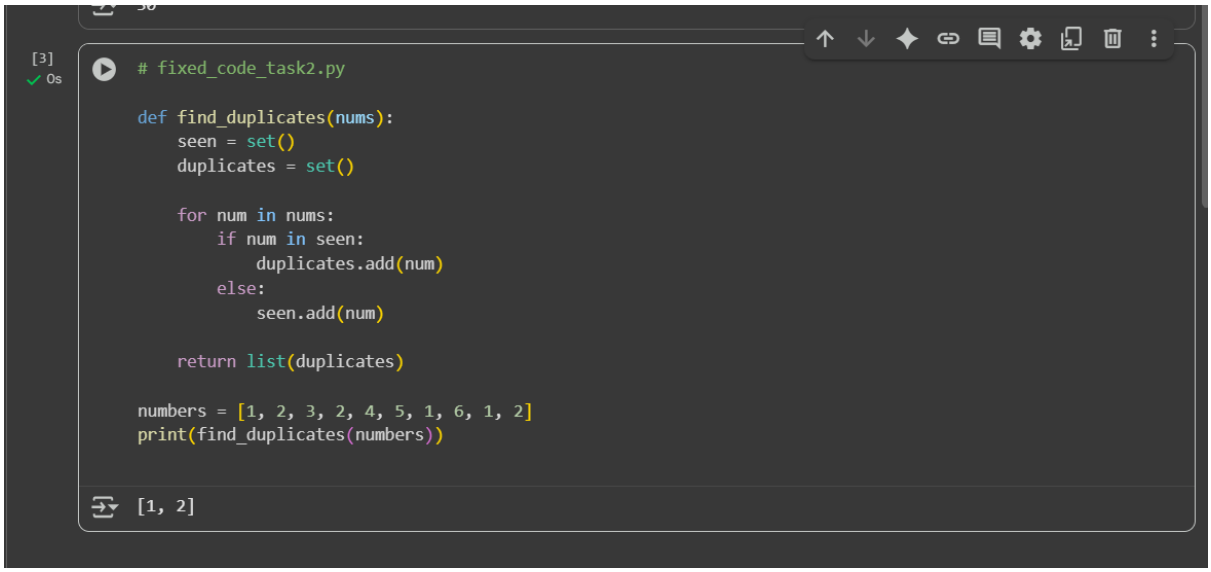
Task 2: Logical and Performance Issue Review

Task: Optimize inefficient logic while keeping the result correct.

```
# buggy_code_task2.py
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:



Optimizations:

- Replaced nested loops with **single loop**.
- Used **set membership** (O(1)) instead of repeated list lookups (O(n)).
- Collect duplicates in a **set** (avoids duplicates automatically).
- Converted result back to a list at the end.

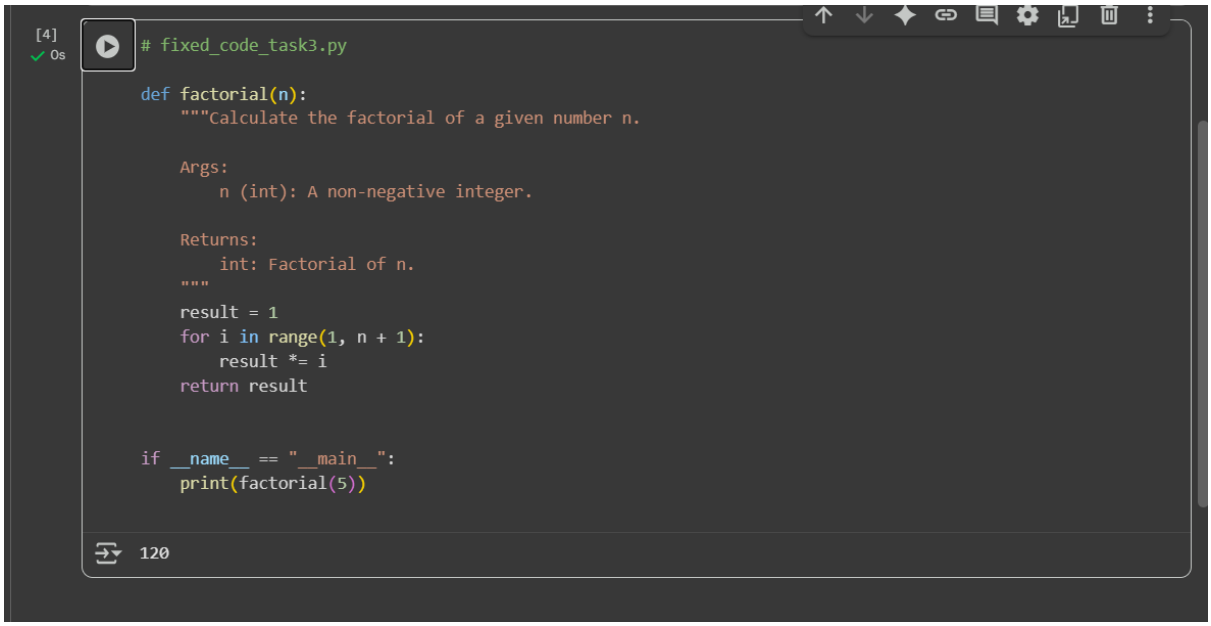
Task 3: Code Refactoring for Readability

Task: Refactor messy code into clean, PEP 8–compliant, well-structured code.

buggy_code_task3.py

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

code:



```
[4]
✓ 0s
# fixed_code_task3.py

def factorial(n):
    """Calculate the factorial of a given number n.

    Args:
        n (int): A non-negative integer.

    Returns:
        int: Factorial of n.
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


if __name__ == "__main__":
    print(factorial(5))

120
```

Refactoring improvements:

- Renamed function c → factorial (clearer).
- Renamed variable x → result.
- Added **docstring** (PEP 257).
- Fixed indentation (4 spaces).
- Wrapped print in if __name__ == "__main__": → better structure for reusability.

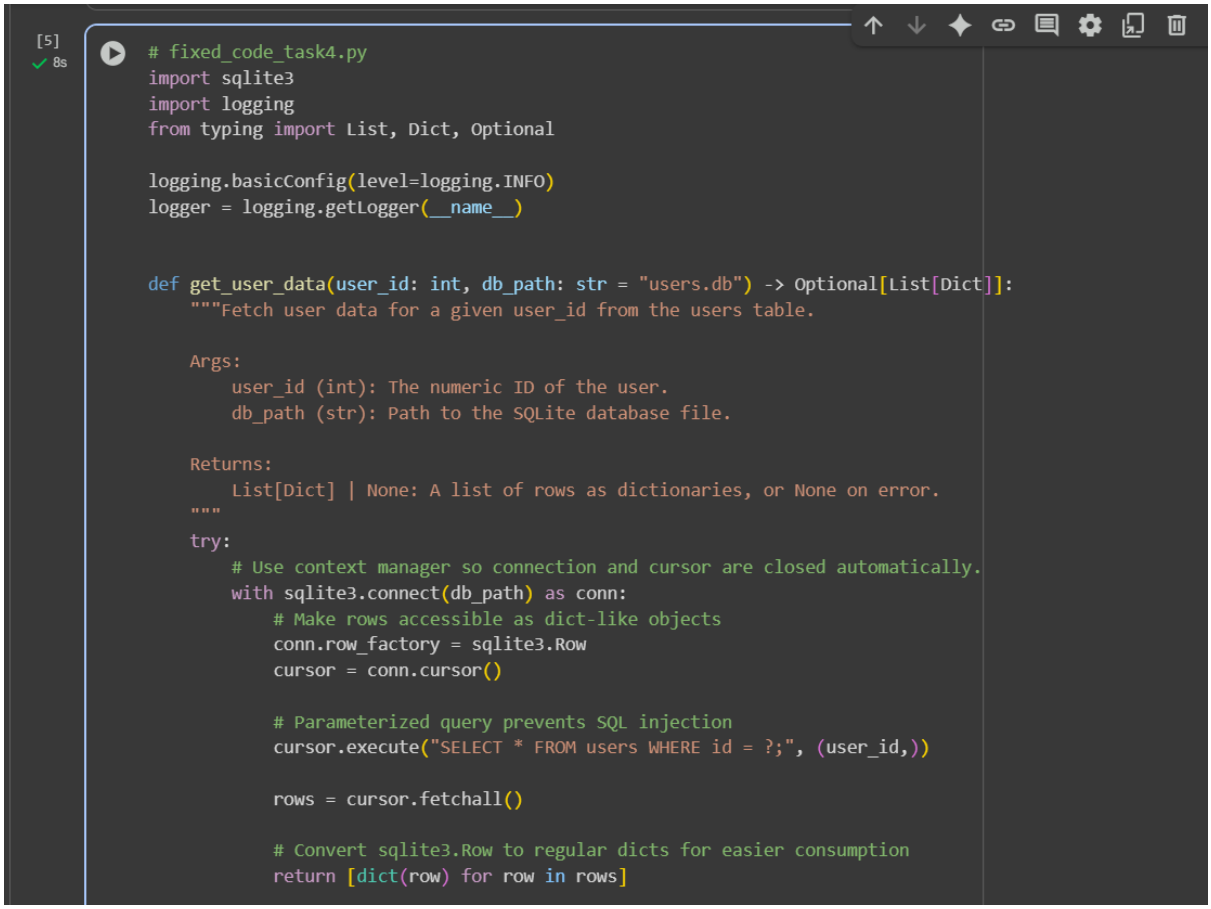
Task 4: Security and Error Handling Enhancement

Task: Add security practices and exception handling to the code.

```
# buggy_code_task4.py
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:



```
[5]
✓ 8s
# fixed_code_task4.py
import sqlite3
import logging
from typing import List, Dict, Optional

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def get_user_data(user_id: int, db_path: str = "users.db") -> Optional[List[Dict]]:
    """Fetch user data for a given user_id from the users table.

    Args:
        user_id (int): The numeric ID of the user.
        db_path (str): Path to the SQLite database file.

    Returns:
        List[Dict] | None: A list of rows as dictionaries, or None on error.
    """
    try:
        # Use context manager so connection and cursor are closed automatically.
        with sqlite3.connect(db_path) as conn:
            # Make rows accessible as dict-like objects
            conn.row_factory = sqlite3.Row
            cursor = conn.cursor()

            # Parameterized query prevents SQL injection
            cursor.execute("SELECT * FROM users WHERE id = ?;", (user_id,))

            rows = cursor.fetchall()

            # Convert sqlite3.Row to regular dicts for easier consumption
            return [dict(row) for row in rows]
```

```
    except sqlite3.Error as e:
        # Log the error for diagnostics; do not expose DB internals to users
        logger.exception("Database error while fetching user data.")
        return None


def prompt_user_id() -> Optional[int]:
    """Prompt user for an integer user ID and return it or None on invalid input."""
    raw = input("Enter user ID: ").strip()
    if not raw:
        logger.info("No input provided.")
        return None

    try:
        uid = int(raw)
        if uid < 0:
            logger.info("User ID must be non-negative.")
            return None
        return uid
    except ValueError:
        logger.info("Invalid user ID. Please enter a numeric value.")
        return None


def main():
    user_id = prompt_user_id()
    if user_id is None:
        print("Invalid input. Exiting.")
        return

    result = get_user_data(user_id)
    if result is None:
        print("An error occurred while fetching user data. Please try again later.")
    elif not result:
        print("No user found with the specified ID.")
```

[5] ✓ 8s

```
        result = get_user_data(user_id)
        if result is None:
            print("An error occurred while fetching user data. Please try again later.")
        elif not result:
            print("No user found with the specified ID.")
        else:
            # Pretty-print the returned records
            for record in result:
                print(record)


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

```
Enter user ID: 2402
ERROR:__main__:Database error while fetching user data.
Traceback (most recent call last):
  File "/tmp/ipython-input-4041158386.py", line 28, in get_user_data
    cursor.execute("SELECT * FROM users WHERE id = ?;", (user_id,))
sqlite3.OperationalError: no such table: users
An error occurred while fetching user data. Please try again later.
```

Code explanation:

 Uses **parameterized queries** (prevents SQL injection).

 Validates and converts the user_id input to an int.

 Uses **context managers** (with) to ensure DB resources are closed.

 Sets row_factory to sqlite3.Row and returns rows as dictionaries.

 Adds **exception handling** for ValueError and sqlite3.Error.

 Uses logging instead of printing raw exceptions (safer for production).

 Avoids exposing internal error details to the end user

Notes & further improvements

- If you expect multiple ways to identify a user (username, email), add separate, strongly validated paths rather than interpolating them into SQL.
- For production systems consider:
    - Storing DB credentials/config separately (not hardcoded).
    - Using an application-level ORM (e.g., SQLAlchemy) for more complex DB logic and safety.
    - More granular logging levels and possibly masking sensitive fields before logging.
- If users.db or the users table doesn't exist yet, this code will handle the error gracefully and log the exception.

Task 5: Automated Code Review Report Generation

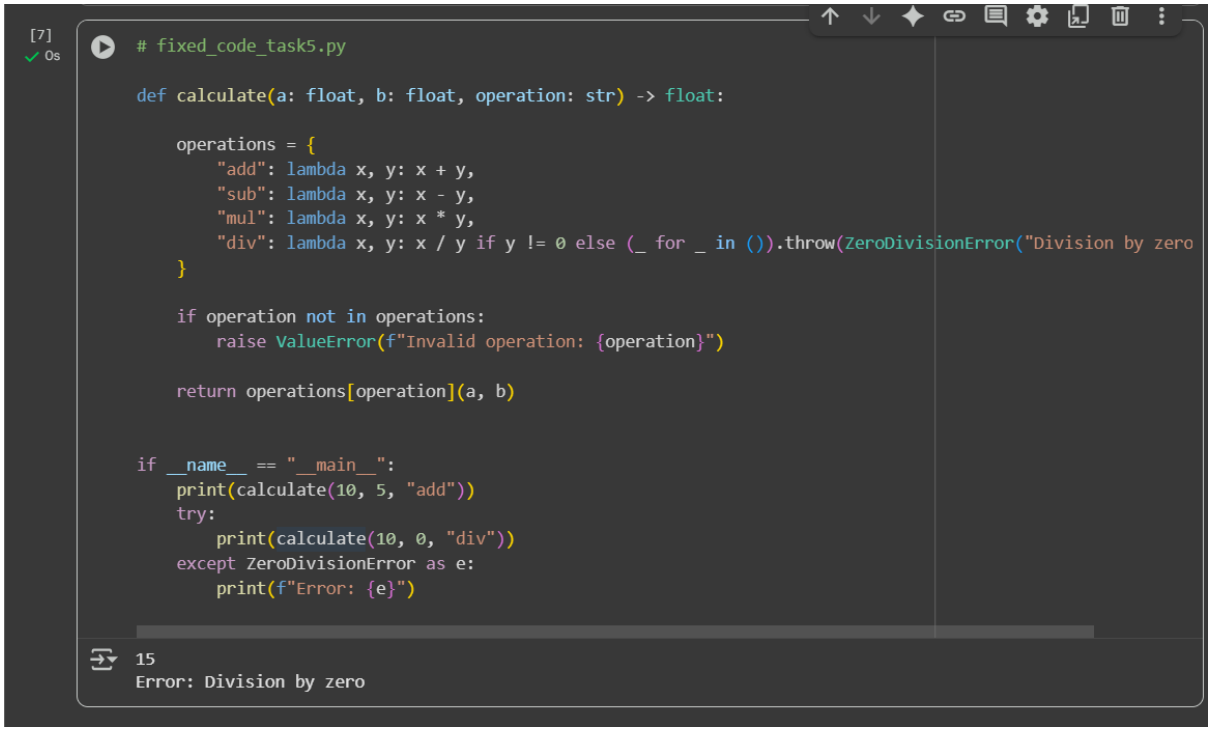
Task: Generate a review report for this messy code.

buggy_code_task5.py

```
def calc(x,y,z):
 if z=="add":
  return x+y
 elif z=="sub": return x-y
 elif z=="mul":
  return x*y
 elif z=="div":
  return x/y
 else: print("wrong")

print(calc(10,5,"add"))
print(calc(10,0,"div"))
```

code:



Code Review Report

File: buggy_code_task5.py

✅ Strengths

1. **Functionality covered:** Supports four basic arithmetic operations (add, subtract, multiply, divide).
2. **Simple and direct structure:** Easy to follow for small use cases.

⚠️ Issues Identified

1. PEP 8 Non-Compliance

- Indentation inconsistent (if and elif blocks should be indented 4 spaces).
- Inline statement elif z=="sub": return x-y breaks readability.
- Function and variable names (calc, x, y, z) are not descriptive.

2. Error Handling

- Division by zero is not handled → causes ZeroDivisionError.
- Invalid operation (else) prints "wrong" but does not return a proper value (returns None).

3. Readability & Maintainability

- No **docstring** explaining the function's purpose, arguments, or return type.
- Operation names ("add", "sub", etc.) are magic strings → prone to typos.
- Mixing of print and return inside the function → inconsistent interface.

4. Scalability

- Every new operation requires another elif.
- Better design would use a **dictionary dispatch pattern** for cleaner extensibility