# AI ASSISTED CODING

LAB 10

## Task 1: Syntax and Error Detection

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

AI.py > lab 10 > ...

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

print(add_numbers(10, 20))
```

1. **Function Definition Syntax**:
   The original code was missing a colon (`:`) at the end of the function definition line. In Python, every function header must end with a colon.
   Fixed:

```python
def add_numbers(a, b):
```

2. **Variable Name Typo**:
   The function tried to return `reslt`, which was a typo. The correct variable name is `result`, which holds the sum of `a` and `b`.
   Fixed:

```python
return result
```

3. **Function Call Arguments**:
   The function call was missing a comma between the two arguments (`10 20`). Arguments in Python must be separated by commas.
   Fixed:

```python
print(add_numbers(10, 20))
```

With these corrections, the code will run successfully and print the sum of 10 and 20.

GPT-4.1 • 1x

## Task 2: Logical and Performance Issue Review

**Task**: Optimize inefficient logic while keeping the result correct.

```python
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))
```

Here's an explanation of the optimized code for finding duplicates:

1. **Original Logic Issue**:
   The original code used nested loops to compare every pair of elements, resulting in $O(n^2)$ time complexity. This is inefficient for large lists.
2. **Optimized Approach**:
   - The code now uses two sets: `seen` to track numbers already encountered, and `duplicates` to store numbers that appear more than once.
   - As it iterates through the list, if a number is already in `seen`, it's added to `duplicates`. Otherwise, it's added to `seen`.
   - This approach ensures each duplicate is only added once and reduces the time complexity to $O(n)$.
3. **Result**:
   The function returns a list of unique duplicate values, maintaining correctness and improving performance.

## Task 3: Code Refactoring for Readability

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

AI.py > lab 10 > ...

```python
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial(5))
```

Here's an explanation of the refactored code:

1. **Function Naming**:
   The function is renamed from `c` to `factorial`, making its purpose clear and descriptive.
2. **Variable Naming**:
   The variable `x` is renamed to `result` for better readability and clarity.
3. **PEP 8 Compliance**:
   - Proper indentation (4 spaces per level).
   - Spaces around operators and after commas.
   - Function and variable names use lowercase letters and underscores.
4. **Structure**:
   The code is organized so that each part is easy to read and understand, following Python's best practices.

This makes the code clean, readable, and maintainable.

## Task 4: Security and Error Handling Enhancement

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

```python
import sqlite3

def get_user_data(user_id):
    try:
        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()
    except sqlite3.Error as e:
        print(f"Database error: {e}")
        result = []
    except Exception as e:
        print(f"Unexpected error: {e}")
        result = []
    finally:
        if 'conn' in locals():
            conn.close()
    return result

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

## Task 4: Security and Error Handling Enhancement

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

```python
import sqlite3

def get_user_data(user_id):
    try:
        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()
    except sqlite3.Error as e:
        print(f"Database error: {e}")
        result = []
    except Exception as e:
        print(f"Unexpected error: {e}")
        result = []
    finally:
        if 'conn' in locals():
            conn.close()
    return result

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

Here's an explanation of the improvements made to the code:

1. **SQL Injection Prevention**:
   The code now uses a parameterized query (`?`) instead of string formatting. This prevents attackers from injecting malicious SQL through user input.
2. **Exception Handling**:
   - The function wraps database operations in a `try-except-finally` block.
   - It catches `sqlite3.Error` for database-specific issues and a general `Exception` for other errors.
   - If an error occurs, it prints a message and returns an empty list.
3. **Resource Management**:
   The database connection is closed in the `finally` block to ensure it always closes, even if an error occurs.
4. **Input Validation**:
   The code checks if the user input is a valid integer before querying the database. If not, it prints an error message.

These changes make the code more secure, robust, and user-friendly.

## Task 5: Automated Code Review Report Generation

**Task**: Generate a **review report** for this messy code.

```python
def review_code():
    report = []
    report.append("Code Review Report for buggy_code_task5.py\n")
    report.append("1. Function Naming:\n- The function name 'calc' is generic. Use descriptive names like 'calculate' or 'arithmetic_operation'.\n")
    report.append("2. Input Validation:\n- No checks for division by zero. This can cause runtime errors.\n")
    report.append("3. Consistency and Readability:\n- Mixed indentation and inconsistent formatting. Use consistent indentation and spacing.\n")
    report.append("4. Error Handling:\n- The function prints 'wrong' for invalid operations but does not return a value. Consider raising exceptions
    report.append("5. PEP 8 Compliance:\n- Use proper spacing, indentation, and avoid one-liners for better readability.\n")
    report.append("6. Usage Example:\n- The code does not handle exceptions when dividing by zero.\n")
    report.append("\nSuggested Improvements:\n")
    report.append("- Use descriptive function and variable names.\n- Add input validation and exception handling.\n- Follow PEP 8 style guidelines.\
    return "".join(report)

print(review_code())
```

Here's an explanation of the review report code:

- The function `review_code()` builds a list of comments about the messy code, covering naming, input validation, readability, error handling, and PEP 8 compliance.
- It points out specific issues, such as lack of division by zero checks, inconsistent formatting, and poor error handling.
- The report also suggests improvements: using descriptive names, adding validation and exception handling, following style guidelines, and returning error messages instead of printing.
- The function returns the report as a formatted string, which is printed for easy reading.

This approach helps systematically analyze and improve code quality.