

Assignment – 10.5

Name: Mohammed Faisaluddin

HT NO: 2303A52214

Batch: 36

Lab 10 – Code Review and Quality: Using AI to Improve Code Quality and Readability

Lab Objectives:

- Use AI for automated code review and quality enhancement
- Identify and fix syntax, logical, performance, and security issues in Python code
- Improve readability and maintainability through structured refactoring and comments
- Apply prompt engineering for targeted improvements
- Evaluate AI-generated suggestions against **PEP 8 standards** and software engineering best practices

Lab Outcomes:

1. Ability to use AI tools to review code
2. Ability to improve code quality and readability
3. Ability to identify and fix common coding issues

Task Description #1 – Variable Naming Issues

Given Code:


```
def f(a, b):  
    return a + b  
  
print(f(10, 20))
```

30

Issues Identified:

- Function name f is unclear
- Variable names a and b lack meaning
- Poor readability

Improved Code:

```
[2]  
✓ Os  def add_numbers(first_number, second_number):  
    """Return the sum of two numbers."""  
    return first_number + second_number  
  
    print(add_numbers(10, 20))
```

... 30

Improvements:

- Meaningful function and variable names
- Added docstring for clarity
- Improved readability and maintainability

Task Description #2 – Missing Error Handling

Given Code:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b

print(divide(10, 0))

***
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-4071441399.py in <cell line: 0>()
      4     return a / b
      5
----> 6 print(divide(10, 0))

/tmp/ipython-input-4071441399.py in divide(a, b)
      1 def divide(a, b):
      2     if b == 0:
----> 3         raise ValueError("Cannot divide by zero!")
      4     return a / b
      5

ValueError: Cannot divide by zero!
```

Next steps: [Explain error](#)

Issues Identified:

- No error handling for division by zero
- Program crashes at runtime

Improved Code:

```
def divide_numbers(numerator, denominator):
    """Divide two numbers safely with error handling."""
    try:
        return numerator / denominator
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    except TypeError:
        return "Error: Please provide numeric values."

print(divide_numbers(10, 0))

*** Error: Division by zero is not allowed.
```

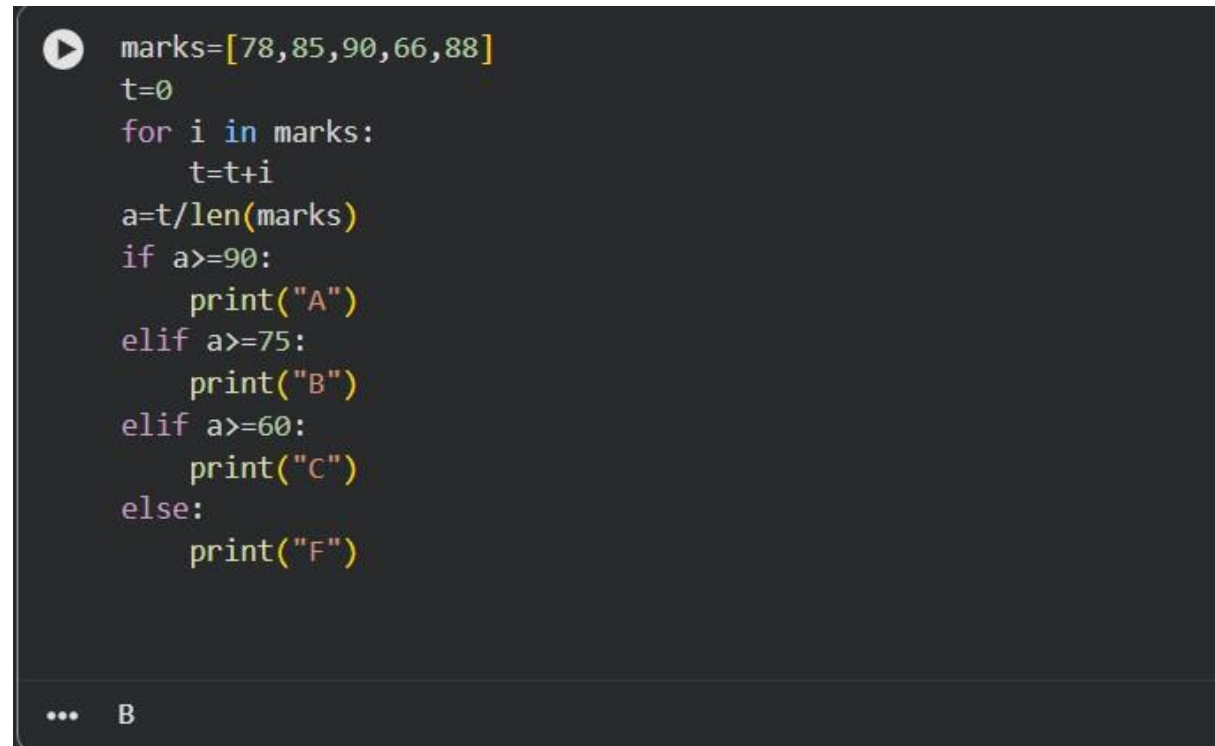
Improvements:

- Added exception handling
- Clear error messages

- Prevents runtime crashes

Task Description #3 – Student Marks Processing System

Given Code:

A screenshot of a code editor with a dark background. The code is written in Python and calculates the average of a list of marks, then prints a grade based on the average. The code is as follows:

```
marks=[78,85,90,66,88]
t=0
for i in marks:
    t=t+i
a=t/len(marks)
if a>=90:
    print("A")
elif a>=75:
    print("B")
elif a>=60:
    print("C")
else:
    print("F")
```

At the bottom left of the editor, there are three dots and the letter 'B'.

Issues Identified:

- Poor variable names
- No functions
- No validation
- Not PEP 8 compliant

Refactored Code:

```
def calculate_grade(marks):  
    """  
    Calculate total marks, average, and grade.  
    Args:  
        marks (list): List of student marks  
    """  
    if not marks:  
        print("Error: Marks list cannot be empty.")  
        return  
  
    total_marks = sum(marks)  
    average_marks = total_marks / len(marks)  
  
    if average_marks >= 90:  
        grade = "A"  
    elif average_marks >= 75:  
        grade = "B"  
    elif average_marks >= 60:  
        grade = "C"  
    else:  
        grade = "F"  
  
    print(f"Total Marks: {total_marks}")  
    print(f"Average Marks: {average_marks:.2f}")  
    print(f"Grade: {grade}")  
  
student_marks = [78, 85, 90, 66, 88]  
calculate_grade(student_marks)  
  
... Total Marks: 407  
Average Marks: 81.40  
Grade: B
```

Improvements:

- Follows PEP 8 standards
- Meaningful variable names
- Modular function-based design
- Added documentation and validation

Task Description #4 – Add Docstrings and Inline Comments

Improved Code:

```
def factorial(n):  
    """  
    Calculate the factorial of a given number.  
  
    Args:  
        n (int): A non-negative integer  
  
    Returns:  
        int: Factorial of the number  
    """  
    result = 1  
  
    # Loop from 1 to n and multiply each value  
    for i in range(1, n + 1):  
        result *= i  
  
    return result
```

Improvements:

- Clear docstring explaining purpose, parameters, and return value
- Inline comments for better understanding

Task Description #5 – Password Validation System (Enhanced)

Original Code:

```
pwd = input("Enter password: ")  
if len(pwd) >= 8:  
    print("Strong")  
else:  
    print("Weak")
```

```
... Enter password: kjv1knkhv  
Strong
```

Limitations:

- Checks only password length
- No real security

- Not reusable
- Poor readability

Enhanced Code:

```
import re

def validate_password(password):
    """
    Validate password based on security rules.
    """
    if len(password) < 8:
        return "Weak: Password must be at least 8 characters long."

    if not re.search(r"[A-Z]", password):
        return "Weak: Must include an uppercase letter."

    if not re.search(r"[a-z]", password):
        return "Weak: Must include a lowercase letter."

    if not re.search(r"[0-9]", password):
        return "Weak: Must include a digit."

    if not re.search(r"[!@#$%^&*()_+=-]", password):
        return "Weak: Must include a special character."

    return "Strong Password"

user_password = input("Enter password: ")
print(validate_password(user_password))
|

... Enter password: jhvjnkpnhgcx
Weak: Must include an uppercase letter.
```

Comparison and Analysis

1. Code Readability and Structure

- Clear function-based design
- Meaningful variable names
- Clean and readable logic

2. Maintainability and Reusability

- Password validation logic is reusable
- Easy to modify or extend security rules

3. Security Strength and Robustness

- Enforces industry-standard password rules

- Reduces risk of weak passwords

Justification of AI-Generated Improvements

- Length check ensures baseline security
- Uppercase and lowercase checks prevent predictable passwords
- Digit requirement increases complexity
- Special characters reduce brute-force success
- Refactoring improves clarity, testability, and maintainability

Conclusion

This lab demonstrated how AI-assisted code review can significantly improve:

- Code readability
- Error handling
- Security
- Maintainability
- Compliance with PEP 8 standards

The refactored programs are safer, cleaner, and suitable for real-world applications.