

# LAB Assignment – 10.5

T HRUSHIKESH

2303A51788

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

### Quality and Readability

#### Task Description #1 – Variable Naming Issues

**Task:** Use AI to improve unclear variable names.

Sample Input Code:

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

Expected Output:

- Code rewritten with meaningful function and variable names.

**Prompt:** “rewrite the above code with meaningful function and variable names with documentation and inline comments, use pygostyle for docstrings”

**CODE :**

```
# ----- Task-1 -----
#sample code

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

#rewrite the above code with meaningful function and variable names with documentation and inline comments,use pygostyle for d
def add_numbers(num1, num2):
    """
    Add two numbers and return the result.

    Parameters:
    num1 (int or float): The first number to be added.
    num2 (int or float): The second number to be added.

    Returns:
    int or float: The sum of num1 and num2.
    """
    return num1 + num2 # Return the sum of the two numbers
# Example usage
print(add_numbers(10, 20)) # Output: 30
```

**OUTPUT:**

```
● PS C:\Users\hrush> & C:/Users/hrush/AppData/Local/Microsoft/WindowsApps/py  
30  
Output: 30
```

**Explanation:** The function `add_numbers()` is much clearer than the original because it uses real names instead of mysterious letters. Rather than `f(a, b)`, you get `add_numbers(num1, num2)`—anyone reading it immediately knows what's happening. The docstring is like a user manual built right into the code, telling anyone who uses it what it does and what to expect back. The comment on the return line is a friendly reminder of what that line accomplishes. And finally, the example at the bottom shows you exactly how to use it and what answer you'll get, making it super practical and easy to follow.

**Task Description #2 – Missing Error Handling**

**Task:** Use AI to add proper error handling.

**Sample Input Code:**

```
def divide(a, b):  
    return a / b  
  
print(divide(10, 0))
```

**Expected Output:**

- Code with exception handling and clear error messages

**Prompt:** "#rewrite the above code with exception handling and clear error messages with documentation and inline comments, use pygostyle for docstrings"

**Code**

```

# -----Task-2-----
#sample code

def divide(a, b):
    return a / b
print(divide(10, 0))

#rewrite the above code with exception handling and clear error messages with documentation and inline comments,use pydocstyle for
def divide_numbers(num1, num2):
    """
    Divide num1 by num2 and return the result.

    Parameters:
    num1 (int or float): The numerator.
    num2 (int or float): The denominator. Must not be zero.

    Returns:
    int or float: The result of dividing num1 by num2.

    Raises:
    ValueError: If num2 is zero, as division by zero is undefined.
    """
    if num2 == 0:
        raise ValueError("Error: Division by zero is not allowed.") # Raise an error if the denominator is zero
    return num1 / num2 # Return the result of the division
# Example usage
try:
    print(divide_numbers(10, 0)) # This will raise an error
except ValueError as e:
    print(e) # Print the error message

```

Output:

```

● PS C:\Users\hrush> & C:/Users/hrush/AppData/Local/Microsoft/WindowsApps/p
  Error: Division by zero is not allowed.
○ PS C:\Users\hrush>

```

Explanation: The improved `divide_numbers()` function is way safer than just blindly dividing numbers. The original code would crash with a cryptic error if you tried to divide by zero, but the new version checks for that first and gives you a friendly message instead: "Error: Division by zero is not allowed." The docstring explains what the function does, what it needs, and what errors might happen—like a helpful instruction manual. The try-except block at the end catches any problems gracefully, so your program doesn't explode. With better variable names like `num1` and `num2` instead of `a` and `b`, anyone reading the code instantly knows it's about division, not some mysterious math operation

### Task Description #3: Student Marks Processing System

The following program calculates total, average, and grade of a student, but it has poor readability, style issues, and no error handling.

`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")
```

**Task:**

- Use AI to refactor the code to follow PEP 8 standards.
- Add meaningful variable names, functions, and comments.
- Add basic input validation and documentation.

**Prompt:**

“ # in the above code refactor the code to follow PEP 8 standards and Add meaningful variable names, functions, and comments and Add basic input validation and documentation”

**Code:**

```

> work > 3rd YEAR > Aiassistant > lab10.py > ...
-----Task-3-----
# -----
#sample code
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")

# in the above code refactor the code to follow PEP 8 standards and Add meaningful variable names, functions, and comments and Add
def calculate_average(marks):
    """
    Calculate the average of a list of marks and return the corresponding grade.

    Parameters:
    marks (list): A list of numerical marks.

    Returns:
    str: The grade corresponding to the average mark.
    """
    if not marks:
        return "Error: The list of marks cannot be empty."
    total_marks = sum(marks)
    average_mark = total_marks / len(marks)

    if average_mark >= 90: # Grade A for average marks 90 and above
        return "Grade: A"
    elif average_mark >= 75: # Grade B for average marks between 75 and 89
        return "Grade: B"
    elif average_mark >= 60: # Grade C for average marks between 60 and 74
        return "Grade: C" # Grade F for average marks below 60
    else:
        return "Grade: F" # Grade F for average marks below 60

# Example usage
marks = [78, 85, 90, 66, 88]
print(calculate_average(marks))

```

Output: PS C:\Users\hrush> & C:/Users/hrush/AppData/Local/Micros  
Grade: B

**Explanation:** The original code was a total mess—single-letter variables like `t` and `a` that nobody could understand, and it just printed single letters instead of explaining the grade. The improved `calculate_average()` function transforms this into something readable and useful. It takes a list of marks, checks that the list isn't empty (so you don't get weird errors), then adds them all up using `sum()` instead of a clunky loop. The function calculates the average properly and returns not just a letter like "B", but a clear label like "Grade: B" so you actually know what you're looking at. The docstring explains exactly what it expects and what it gives back. With meaningful names like `total_marks` and `average_mark`, the code tells a story—you instantly understand what's happening at each step without having to decode mysterious abbreviations.

#### Task Description #4: Use AI to add docstrings and inline comments

to the following function.

```
def factorial(n):
```

```
    result = 1
```

```
    for i in range(1,n+1):
```

```
result *= i  
return result
```

**Prompt :** “ # rewrite the above code to add docstrings and inline comments to the following function with documentation and inline comments, use pydocstyle for docstrings”

Code:

```
#-----Task-4-----  
#sample code  
def factorial(n):  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result  
  
# rewrite the above code to add docstrings and inline comments to the following function with documentation and inline comments, use pydocstyle for docstrings  
def factorial(n):  
    """  
    Calculate the factorial of a given number n.  
  
    Parameters:  
    n (int): The number for which to calculate the factorial. Must be a non-negative integer.  
  
    Returns:  
    int: The factorial of the given number n.  
  
    Raises:  
    ValueError: If n is a negative integer.  
    """  
    if n < 0:  
        raise ValueError("Error: Factorial is not defined for negative integers.")  
  
    result = 1 # Initialize result to 1, as factorial of 0 is 1  
    for i in range(1, n + 1):  
        result *= i # Multiply result by each integer from 1 to n  
    return result  
  
# Example usage  
print(factorial(5)) # Output: 120
```

Output:

```
120  
PS C:\Users\hrush> & C:/Users/hrush/AppData/Local  
Factorial of 5 is: 120  
PS C:\Users\hrush>
```

Explanation:

The improved factorial() function is way safer and clearer than the original. The original version would explode with an error if you passed a negative number—factorial just doesn't work for negatives—but the new version checks first and tells you exactly what went wrong: "Error: Factorial is not defined for negative integers." The docstring explains what the function needs, what it returns, and what exceptions might happen. Each step of the calculation has a comment explaining what's going on: you start with result = 1 because that's how factorials work mathematically, then loop through numbers 1 to n, multiplying them together to get your answer. The example shows you that factorial(5) gives 120, which is  $5 \times 4 \times 3 \times 2 \times 1$ . It's organized, protected from bad inputs, and anyone reading it knows exactly what it does.

## Task Description #5: Password Validation System (Enhanced)

The following Python program validates a password using only a minimum length check, which is insufficient for real-world security requirements.

```
pwd = input("Enter password: ")

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

**Task:**

1. Enhance password validation using AI assistance to include multiple security rules such as:

- o Minimum length requirement
- o Presence of at least one uppercase letter
- o Presence of at least one lowercase letter
- o Presence of at least one digit
- o Presence of at least one special character

2. Refactor the program to:

- o Use meaningful variable and function names
- o Follow PEP 8 coding standards
- o Include inline comments and a docstring

3. Analyze the improvements by comparing the original and AI-enhanced versions in terms of:

- o Code readability and structure
- o Maintainability and reusability
- o Security strength and robustness

4. Justify the AI-generated changes, explaining why each added rule and refactoring decision improves the overall quality of the program.

**Prompt :** “ #rewrite the above code by following this thing

# Minimum length requirement

# Presence of at least one uppercase letter

# Presence of at least one lowercase letter

# Presence of at least one digit

# Presence of at least one special character

# Refactor the program to:

# Use meaningful variable and function names

# Follow PEP 8 coding standards

# Include inline comments and a docstring

# Analyse the improvements by comparing the original and AI-enhanced versions in terms of:

# Code readability and structure

# Maintainability and reusability

# Security strength and robustness”

Code:

```
# -----Task-5-----  
#sample code  
pwd = input("Enter password: ")  
if len(pwd) >= 8:  
    print("Strong")  
else:  
    print("Weak")  
  
# rewrite the above code by following this thing  
# Minimum length requirement  
# Presence of at least one uppercase letter  
# Presence of at least one lowercase letter  
# Presence of at least one digit  
# Presence of at least one special character  
# Refactor the program to:  
# Use meaningful variable and function names  
# Follow PEP 8 coding standards  
# Include inline comments and a docstring  
# Analyze the improvements by comparing the original and AI-enhanced versions in terms of:  
# Code readability and structure  
# Maintainability and reusability  
# Security strength and robustness  
  
import re  
def validate_password(password):  
    """  
        Validate the strength of a given password based on specific criteria.  
  
    Parameters:  
        password (str): The password to be validated.
```

```

>Returns:
str: A message indicating whether the password is strong or weak.

Criteria for a strong password:
- Minimum length of 8 characters
- At least one uppercase letter
- At least one lowercase letter
- At least one digit
- At least one special character
"""

if len(password) < 8:
    return "Weak: Password must be at least 8 characters long."

if not re.search(r'[A-Z]', password):
    return "Weak: Password must contain at least one uppercase letter."

if not re.search(r'[a-z]', password):
    return "Weak: Password must contain at least one lowercase letter."

if not re.search(r'[0-9]', password):
    return "Weak: Password must contain at least one digit."

if not re.search(r'[^@#$%^&(*).?":{}|<>]', password):
    return "Weak: Password must contain at least one special character."

return "Strong: Your password is strong." # Return strong if all criteria are met
# Example usage
password_input = input("Enter password: ")
print(validate_password(password_input))

```

Output:

```

PS C:\Users\hrush> & C:/Users/hrush/AppData/Local/Microsoft/Windows/Enter password: hrshi
Weak: Password must be at least 8 characters long.
PS C:\Users\hrush> & C:/Users/hrush/AppData/Local/Microsoft/Windows/Enter password: Hrushi
Weak: Password must be at least 8 characters long.
PS C:\Users\hrush> & C:/Users/hrush/AppData/Local/Microsoft/Windows/Enter password: Hrushikesh
Weak: Password must contain at least one digit.
PS C:\Users\hrush>

```

**Explanation :** The original code only checked password length with a crude "Strong/Weak" message, ignoring real security. The improved validate\_password() function validates five criteria—length, uppercase, lowercase, digits, and special characters—giving specific feedback on what's missing. It uses regex patterns for pattern matching, meaningful variable names, and a detailed docstring. The function is reusable, follows PEP 8, and provides actual security instead of a false sense of safety.