

AI ASSISTED CODING

S.Rishaak

2303A51125

BATCH - 03

20 - 02 - 2026

ASSIGNMENT – 10.5

LAB – 10.5 : Code Review and Quality : Using AI to improve code Quality and Readability.

Task – 01: Variable Naming Issues.

Prompt: Review the following Python code and improve it by replacing unclear function and variable names with meaningful and descriptive names. Refactor the code to follow PEP 8 standards and improve readability and maintainability without changing its functionality.

Code & Output:

The screenshot shows a code editor interface with two sections: 'ORIGINAL CODE' and 'IMPROVED CODE'. In the 'ORIGINAL CODE' section, the code is:[1] def f(a, b):
 return a + b
print(f(10, 20))
... 30In the 'IMPROVED CODE' section, the code is refactored:[2] def add_numbers(first_number, second_number):
 return first_number + second_number
result = add_numbers(10, 20)
print(result)
... 30The code editor has a toolbar at the top with File, Edit, View, Insert, Runtime, Tools, Help, and a search bar for Commands. It also includes tabs for Code, Text, Run all, RAM, and Disk. At the bottom, there are buttons for Variables, Terminal, and a blue run button. The status bar shows 11:26 AM and Python 3.

Explanation :

The original code used unclear function and variable names, making it difficult to understand its purpose. AI improved the code by using meaningful names and adding structure, which enhances readability and maintainability.

Task – 02 : Missing Error Handling.

Prompt: Review the following Python code and improve it by adding proper error handling. Handle possible exceptions such as division by zero and invalid input types. Refactor the code to follow PEP 8 standards, use meaningful variable names, and provide clear, user-friendly error messages without changing the core functionality.

Code & Output :

The screenshot shows a Jupyter Notebook interface with the title "TASK 02". Under the "ORIGINAL CODE" section, there is a cell containing the following Python code:

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

...
ZeroDivisionError: division by zero
```

When run, it produces the following traceback:

```
/tmp/ipython-input-4288846623.py in <cell line: 0>()
      1 def divide(a, b):
      2     return a / b
----> 3 print(divide(10, 0))

/tmp/ipython-input-4288846623.py in divide(a, b)
      1 def divide(a, b):
      2     return a / b
----> 3 print(divide(10, 0))

ZeroDivisionError: division by zero
```

The status bar at the bottom indicates "11:44 AM" and "Python 3".

The screenshot shows a Jupyter Notebook interface with the title "IMPROVED CODE". The code in the cell is as follows:

```
[4]: def divide_numbers(dividend, divisor):
    try:
        return dividend / divisor
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    except TypeError:
        return "Error: Invalid input type. Please enter numbers only."
    result = divide_numbers(10, 0)
    print(result)

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

Explanation :

The original code does not handle runtime errors like division by zero, which can cause the program to crash. The improved version adds exception handling to manage errors gracefully and display clear, user-friendly messages. This enhances program reliability, robustness, and overall code quality.

Task – 03 : Student Marks Processing System.

Prompt : Review the following Python program and refactor it to improve readability, structure, and code quality. Follow PEP 8 standards, use meaningful variable and function names, and convert the logic into reusable functions. Add proper input validation, error handling, comments, and a clear docstring. Do not change the core functionality.

Code & Output :

```

[TASK 03]

ORIGINAL CODE
[6] 0s
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")

... B

+ Code + Text

IMPROVED CODE
Variables Terminal 11:53 AM Python 3

```

```

File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all RAM Disk

IMPROVED CODE
[6] 0s
def calculate_grade(marks_list):
    if not marks_list:
        raise ValueError("Marks list cannot be empty.")
    if not all(isinstance(mark, (int, float)) for mark in marks_list):
        raise TypeError("All marks must be numeric values.")
    total_marks = sum(marks_list)
    average_marks = total_marks / len(marks_list)
    if average_marks >= 90:
        grade = "A"
    elif average_marks >= 75:
        grade = "B"
    elif average_marks >= 60:
        grade = "C"
    else:
        grade = "F"
    return total_marks, average_marks, grade
student_marks = [78, 85, 90, 66, 88]
try:
    total, average, grade = calculate_grade(student_marks)
    print(f"Total Marks: ({total})")
    print(f"Average Marks: ({average:.2f})")
    print(f"Grade: {grade}")
except (ValueError, TypeError) as error:
    print(f"Error: {error}")

... Total Marks: 407
Average Marks: 81.40
Grade: B

Variables Terminal 11:53 AM Python 3

```

Explanation :

The original program had poor variable naming, no function structure, and lacked input validation, making it difficult to maintain and understand. The refactored version follows PEP 8 standards, uses meaningful names, and organizes the logic into reusable functions with proper validation. This improves readability, maintainability, and overall code quality.

Task – 04: Use AI to add docstrings and inline comments to the following Function.

Prompt: Review the following Python function and enhance it by adding a proper docstring and meaningful inline comments. Ensure the

documentation explains the purpose, parameters, return value, and possible exceptions. Follow PEP 8 standards and improve readability without changing the core functionality.

Code & Output :

The screenshot shows a code editor interface with two sections: 'ORIGINAL CODE' and 'IMPROVED CODE'. The 'ORIGINAL CODE' section contains a simple factorial function definition. The 'IMPROVED CODE' section contains a more robust version with type and value error handling, a docstring, and inline comments. The code editor has a toolbar at the top, a status bar at the bottom, and tabs for 'Variables' and 'Terminal'.

```
[13] def factorial(n):
    result = 1
    for i in range(1,n+1):
        result *= i
    return result
print(factorial(5))

[14] def factorial(number):
    if not isinstance(number, int):
        raise TypeError("Input must be an integer.")
    if number < 0:
        raise ValueError("Factorial is not defined for negative numbers.")

    result = 1
    for i in range(1, number + 1):
        result *= i
    return result
print(factorial(5))
```

Explanation :

The original function lacked documentation and comments, making it harder to understand its purpose and logic. The improved version adds a clear docstring and inline comments, explaining the functionality, parameters, and return value. This enhances readability, maintainability, and adherence to coding best practices.

Task – 05 : Password Validation System.

Prompt : Refactor the code using meaningful function names, PEP 8 standards, and include a proper docstring with inline comments.

Code & Output :

The screenshot shows a Python code editor interface with two sections: "ORIGINAL CODE" and "IMPROVED CODE".

ORIGINAL CODE:

```
[16] pwd = input("Enter password: ")
if len(pwd) >= 8:
    print("Strong")
else:
    print("Weak")
Enter password: 123456
Weak
```

IMPROVED CODE:

```
[18] import string
def is_password_strong(password):
    if len(password) < 8:
        return False
    has_uppercase = any(char.isupper() for char in password)
```

At the bottom, there are tabs for "Variables" and "Terminal". The status bar shows "12:13PM" and "Python 3".

The screenshot shows a Python code editor interface with one section: "IMPROVED CODE".

```
[18] import string
def is_password_strong(password):
    if len(password) < 8:
        return False
    has_uppercase = any(char.isupper() for char in password)
    has_lowercase = any(char.islower() for char in password)
    has_digit = any(char.isdigit() for char in password)
    has_special = any(char in string.punctuation for char in password)
    return all([has_uppercase, has_lowercase, has_digit, has_special])
def main():
    user_password = input("Enter password: ")
    if is_password_strong(user_password):
        print("Strong Password ✅")
    else:
        print("Weak Password ❌")
if __name__ == "__main__":
    main()
...
... Enter password: 123456
Weak Password ❌
```

At the bottom, there are tabs for "Variables" and "Terminal". The status bar shows "12:13PM" and "Python 3".

Explanation :

Maintainability & Reusability

- Password logic inside a function
- Can reuse in web apps, login systems
- Easy to update rules

Password Security Rules

Rule	Why It Improves Security
Minimum Length	Prevents short brute-force attacks
Uppercase	Increases complexity
Lowercase	Improves character variation
Digit	Adds numeric complexity
Special Character	Maximizes entropy

Maintainability & Reusability

Original	Enhanced
✗ Cannot reuse validation logic	✓ <code>is_password_strong()</code> reusable
✗ Hard to extend	✓ Easy to add new rules
✗ No separation of concerns	✓ Logic separated from input/output

 The enhanced version is more **maintainable** and **scalable**.

Original	Enhanced
Single condition	Multiple structured rules
No function	Modular function
No comments	Docstring + inline comments
Poor naming	Clear naming