ASSIGNMENT 10.5

NAME:THATIKONDA SASYA
Batch:20
HTNO: 2303A51346

TASK 1:

Task: Use AI to improve unclear variable names.

Sample Input Code:

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

```
def add_two_numbers(first_number, second_number):
    return first_number + second_number

result = add_two_numbers(10, 20)
print(result)
```

```
30
```

Justification

Using meaningful function and variable names makes the code **self-descriptive**, reducing the need for additional comments. A name like add_two_numbers immediately conveys the function's purpose, while first_number and second_number clarify the role of each input. This improves **readability** for other developers and for future maintenance. Clear naming also minimizes logical errors and makes debugging easier. Overall, such practices lead to cleaner, more professional, and maintainable code.

TASK-2

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

```
def divide(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    except TypeError:
        return "Error: Both inputs must be numbers."
    except Exception as e:
        return f"Unexpected error occurred: {e}"

print(divide(10, 0))
```

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

Justification:

Error handling is essential to prevent program crashes and ensure reliability during runtime. In the given code, division is a risky operation because it can fail when the divisor is zero or when invalid data types are passed as inputs. By using exception handling, the program can gracefully manage such errors instead of terminating abruptly. Clear and specific error messages help users understand what went wrong and how to correct their input. This improves code robustness, enhances user experience, and makes debugging easier, which is a key objective when applying AI-assisted error handling.

TASK-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.

```python
def calculate_total_and_average(marks):
    """
    Calculates the total and average of student marks.

    Args:
        marks (list): A list of numeric marks.

    Returns:
        tuple: total marks and average marks.

    Raises:
        ValueError: If marks list is empty or contains invalid values.
    """
    if not marks:
        raise ValueError("Marks list cannot be empty.")

    if not all(isinstance(mark, (int, float)) for mark in marks):
        raise ValueError("All marks must be numeric values.")

    total_marks = sum(marks)
    average_marks = total_marks / len(marks)

    return total_marks, average_marks


def determine_grade(average):
    """
    Determines the grade based on average marks.

    Args:
        average (float): Average marks.
```

```python
def determine_grade(average):
    if average >= 90:
        return "A"
    elif average >= 75:
        return "B"
    elif average >= 60:
        return "C"
    else:
        return "F"


def main():
    """
    Main function to process student marks and display grade.
    """
    student_marks = [78, 85, 90, 66, 88]

    try:
        total, average = calculate_total_and_average(student_marks)
        grade = determine_grade(average)

        print(f"Total Marks: {total}")
        print(f"Average Marks: {average:.2f}")
        print(f"Grade: {grade}")

    except ValueError as error:
        print(f"Input Error: {error}")


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

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

Justification:

The original program lacked readability, structure, and safeguards against invalid input, making it difficult to understand and maintain. Refactoring the code according to PEP 8 standards improves clarity through proper indentation, naming conventions, and modular design. Breaking the logic into well-defined functions enhances reusability and simplifies testing. Adding docstrings and comments provides clear documentation for future developers. Basic input validation ensures that errors such as empty or non-numeric marks are handled gracefully, preventing unexpected crashes. Overall, these improvements make the system more robust, maintainable, and suitable for real-world student data processing.

TASK-4:

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

```python
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    Args:
        n (int): A non-negative integer for which to calculate the factorial.

    Returns:
        int: The factorial of n (n!).

    Example:
        >>> factorial(5)
        120
    """
    # Initialize result to 1 (the multiplicative identity)
    result = 1

    # Iterate from 1 to n (inclusive) and multiply each value
    for i in range(1, n + 1):
        result *= i

    # Return the calculated factorial
    return result
```

Justification:

The given function correctly calculates the factorial of a non-negative integer by using an iterative loop. It initializes the result to 1 and multiplies it with each number from 1 to n, following the mathematical definition of factorial. The use of a docstring improves readability and clearly explains the purpose, parameters, and return value of the function. Overall, the code is simple, efficient with O(n) time complexity, and produces accurate results for valid non-negative integer inputs

TASK-5

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

```python
import re


def validate_password(password):
    """
    Validate password strength based on multiple security criteria.

    Args:
        password (str): The password to validate

    Returns:
        tuple: (is_valid, feedback) where is_valid is bool and feedback is str
    """
    # Check minimum length requirement
    if len(password) < 8:
        return False, "Password must be at least 8 characters long."

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

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

```python
    # Check for at least one digit
    if not re.search(r'\d', password):
        return False, "Password must contain at least one digit."

    # Check for at least one special character
    if not re.search(r'[!@#$%^&*()_+\-=\[\]{};:\'",.<>?/\\|`~]', password):
        return False, "Password must contain at least one special character."

    return True, "Strong password"

# Get password input from user
pwd = input("Enter password: ")
is_valid, feedback = validate_password(pwd)
print(feedback)
```

```
Enter password: Alice_4@
Strong password
```

Justification:

This password validation program is designed to enforce strong security practices by checking multiple essential criteria such as minimum length, presence of uppercase and lowercase letters, digits, and special characters. Using regular expressions allows efficient and precise pattern matching for each rule. The function returns both a boolean result and a clear feedback message, making it user-friendly and easy to integrate into larger authentication systems. Clear comments and a descriptive docstring improve readability and maintainability. Overall, the implementation enhances security, prevents weak passwords, and provides meaningful guidance to users for creating strong passwords.