Assignment 10.5

Name: M.karthik

HT NO:2303A51335

Batch:20

Task Description #1 – Variable Naming Issues

Task: Use AI to improve unclear variable names.

Sample Input Code:

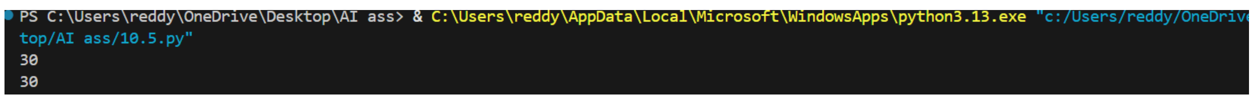
```
def f(a, b):

return a + b

print(f(10, 20))
```

Expected Output:
• Code rewritten with meaningful function and variable names.

Promt: #using ai improve unclear variable names in above code for better readability and understanding.

Code:
```
def add_numbers(num1, num2):

    """Return the sum of num1 and num2."""

    return num1 + num2

print(add_numbers(10, 20))  # Example usage of the add_numbers function
```

Output:



```
PS C:\Users\reddy\OneDrive\Desktop\AI ass> & C:\Users\reddy\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/reddy/OneDrive
top/AI ass/10.5.py"
30
30
```

Observation: The first function works correctly, but the variable names a and b are very short and don't clearly explain what they represent. Someone reading the code might need to guess their meaning. In the improved version, the function name add_numbers clearly tells what the function does. The parameters num1 and num2 also make it obvious that they are numbers being added. Adding a short docstring further improves understanding by explaining the purpose of the function. Overall, the second version is easier to read and more beginner-friendly. This shows how meaningful names and small comments can make code clearer without changing how it works.

Task Description #2 – Missing Error Handling

Task: Use AI to add proper error handling.

Sample Input Code:

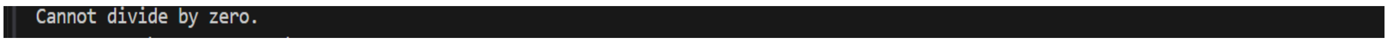
```
def divide(a, b):

return a / b

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

Expected Output:
• Code with exception handling and clear error messages

Promt: #Use AI to add proper error handling.

Code:
```
def divide(a, b):

    """Return the quotient of a and b. Raises ValueError if b is zero."""

    if b == 0:

        raise ValueError("Cannot divide by zero.")

    return a / b

try:

    print(divide(10, 0))

except ValueError as e:

    print(e)  # Output: Cannot divide by zero.
```

Output:



```
Cannot divide by zero.
```

Observation:
The original divide function could crash the program because it tried to divide by zero without checking. This is a common mistake that leads to runtime errors. In the improved version, a condition checks if the second number is zero before dividing. If it is zero, the function raises a clear and meaningful error message. The use of a try-except block also prevents the program from stopping suddenly. Instead, it prints a friendly message to the user. Overall, the updated code is safer, more robust, and easier to understand for beginners.

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

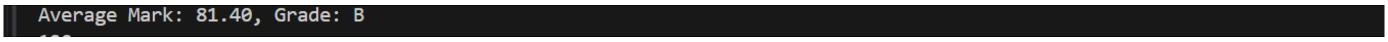
Promt: # Use AI to refactor the code to follow PEP 8 standards.

# Add meaningful variable names, functions, and comments.

# Add basic input validation and documentation.

```
Code:  def calculate_average(marks):
    """Calculate the average of a list of marks.


    Parameters:

    marks (list): A list of numerical marks.


    Returns:

    float: The average of the marks.

    """
    if not marks:
        raise ValueError("The list of marks cannot be empty.")

    total_marks = sum(marks)
    average = total_marks / len(marks)
    return average
def determine_grade(average):
    """Determine the grade based on the average mark.

    Parameters:
    average (float): The average mark.

    Returns:
    str: The grade corresponding to the average mark.
    """
    if average >= 90:
        return "A"
    elif average >= 75:
        return "B"
    elif average >= 60:
        return "C"
    else:
        return "F"
# Example usage:
marks = [78, 85, 90, 66, 88]
try:
    average_mark = calculate_average(marks)
    grade = determine_grade(average_mark)
    print(f"Average Mark: {average_mark:.2f}, Grade: {grade}")
except ValueError as e:
    print(e)  # Output: The list of marks cannot be empty.
```

Output:



```
Average Mark: 81.40, Grade: B
```

Observation:
The original code was difficult to read because it had no proper indentation, unclear variable names like t and a, and all logic written in one block. This makes it hard to understand and maintain. In the improved version, the logic is split into clear functions such as calculate_average and determine_grade, which makes the purpose obvious. Meaningful variable names and comments help readers quickly understand what the code is doing. Input validation also checks if the marks list is empty, preventing possible errors. Proper formatting and structure follow Python standards, making the code cleaner. Overall, the refactored code is more organized, readable, and beginner-friendly.

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

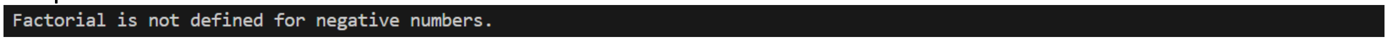
to the following function.

```
def factorial(n):

result = 1

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

result *= i
return result
```

Prompt: #Use AI to add docstrings and inline comments to the following function.

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


    Parameters:

    n (int): A non-negative integer whose factorial is to be calculated.


    Returns:

    int: The factorial of n.


    Raises:

    ValueError: If n is negative.
    """

    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")

    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:
try:
    print(factorial(5))  # Output: 120
    print(factorial(-1)) # This will raise an error
except ValueError as e:
    print(e)  # Output: Factorial is not defined for negative numbers.
```

Output:



```
Factorial is not defined for negative numbers.
```

OBSERVATION :

The original factorial code had no indentation, comments, or explanation, so it was hard to follow what was happening. In the improved version, a clear docstring explains what the function does, its input, output, and possible errors. Inline comments describe each important step, which helps beginners understand the logic easily. The function also checks for negative numbers and raises a proper error instead of giving wrong results. This makes the code more reliable and user-friendly. The example usage with try-except shows how to safely call the function. Overall, the updated code is well-documented, clear, and easy to learn from.

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: # Enhance password validation using AI assistance to includemultiple security rules such as

# 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

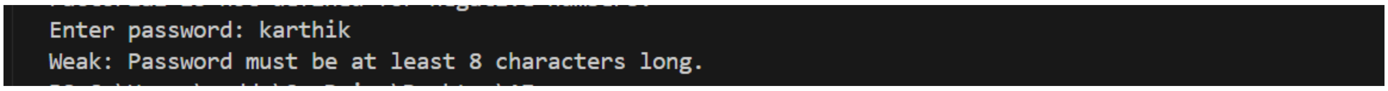
# Security strength and robustness

# Justify the AI-generated changes, explaining why each addedrule and refactoring decision improves the overall quality of the program.

Code: import re

```
def validate_password(password):
    """Validate the strength of a password based on multiple security rules.


    Parameters:
    password (str): The password to be validated.


    Returns:
    str: A message indicating whether the password is strong or weak.
    """
    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"  # If all conditions are met, the password is strong
# Example usage:
password_input = input("Enter password: ")
validation_result = validate_password(password_input)
print(validation_result)  # Output will indicate whether the password is strong or weak based on the validation rules.
```

Output:



```
Enter password: karthik
Weak: Password must be at least 8 characters long.
```

OBSERVATION :

The original code only checked the length of the password, so it was very basic and not secure. The improved version adds multiple checks like uppercase, lowercase, digits, and special characters, which makes the password much stronger. The logic is moved into a function called validate_password, making the code more organized and reusable. Clear variable names and proper formatting improve readability and follow Python standards. Comments and a docstring explain the purpose and rules, helping others understand and maintain the code. Using regular expressions makes the validation accurate and flexible. Overall, the AI-enhanced version is cleaner, safer, and more professional than the original simple check.