ASSIGNMENT- 10.5

Name: Akhila Cherepally

HT.No: 2303A51417

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

PROMPT :

#using ai improve the unclear variable names in above code

CODE:

```
#TASK 1
def f(a,b):
    return a+b
print(f(10,20))
#using ai improve the unclear variable names in above code
def add_numbers(num1, num2):
    return num1 + num2
print(add_numbers(10, 20))
```

OUTPUT:

```
PS C:\Users\akhil\OneDrive\Desktop\Ai asst Coding> & C:\Users\akhil\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/User
/Assignment 10.5.py"
30
30
PS C:\Users\akhil\OneDrive\Desktop\Ai asst Coding> & C:\Users\akhil\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/User
```

OBSERVATION:

The improved version of the code is much clearer and easier to understand. In the first function, the variable names a and b are very short and do not clearly explain what values they represent. After renaming them to num1 and num2, the purpose of the function becomes obvious. The function name add_numbers is also more meaningful compared to just f, which makes the code more readable.

Using descriptive variable and function names improves clarity and makes the program easier to maintain. It also helps others quickly understand what the code is doing without extra explanation. This follows good coding practices and improves overall code quality while keeping the same functionality.

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:

#using ai improve the above code to add proper error handling with exception handling

CODE:

```
def divide(num1, num2):
    try:
        result = num1 / num2
        return result
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
print(divide(10, 0))
```

OUTPUT:

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

OBSERVATION:

The improved version of the code is more reliable and safer to use. In the original version, dividing by zero would cause the program to crash. By adding exception handling with try and except, the error is handled properly without stopping the program.
The use of ZeroDivisionError makes the code more specific and meaningful, as it clearly addresses the exact issue. Returning a proper error message instead of letting the program fail improves user experience. The variable names num1 and num2 also make the function easier to understand. Overall, the updated version follows good programming practices by making the code more stable and user-friendly.

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:

#using ai refactor the code to follow pep8 standards . add meaningful variable names , functions and comments , add basic input validation and documentation

CODE:

```
def calculate_average_grade(marks):
    """
    Calculate the average grade based on the provided marks and return the corresponding letter grade.

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

    Returns:
    str: The letter grade corresponding to the average mark.
    """
    if not marks:
        return "Error: No marks provided."

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

    if average_mark >= 90:
        return "A"
    elif average_mark >= 75:
        return "B"
    elif average_mark >= 60:
        return "C"
    else:
```

```
        return "F"
# Example usage
marks = [78, 85, 90, 66, 88]
grade = calculate_average_grade(marks)
print(f"The average grade is: {grade}")
```

OUTPUT:

```
The average grade is: B
```

OBSERVATION:

The refactored version of the code is more structured and easier to understand. The logic has been placed inside a function, which improves reusability and keeps the program organized. Meaningful names like calculate_average_grade, total_marks, and average_mark clearly describe their purpose, making the code more readable.

The addition of a proper docstring explains what the function does, its parameters, and its return value, which follows good documentation practices. Basic input validation is included by checking if the marks list is empty, preventing potential errors. The use of built-in functions like sum() also makes the code cleaner and more efficient. Overall, the refactored version follows PEP 8 standards and improves clarity, maintainability, and reliability.

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:

```
# using ai add docstrings and inline comments to the above code
```

CODE:

```
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 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
try:
    print(factorial(5))  # Output: 120
    print(factorial(-3)) # This will raise an error
except ValueError as e:
    print(e)
```

OUTPUT:

```
120
Error: Factorial is not defined for negative integers.
PS C:\Users\akhil\OneDrive\Desktop\Ai asst Coding> & C:\Users\akhil\Ap
```

OBSERVATION:

The updated version of the factorial code is much clearer and more complete. A proper docstring is added, which explains what the function does, what input it expects, and what it returns. This makes the function easier to understand for anyone reading the code.

Inline comments are also included to explain important steps, like why the result is initialized to 1 and what happens inside the loop. Error handling for negative numbers makes the function safer and more reliable. The example usage with try and except shows how the function behaves in both normal and error cases. Overall, the code is more readable, well-documented, and follows good programming practices.

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

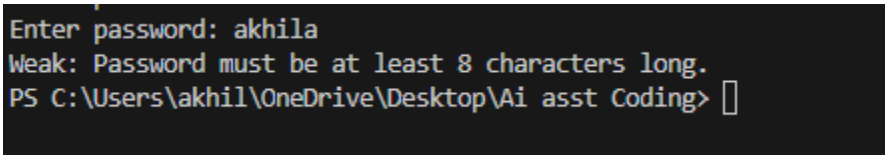
#Enhance password validation using AI assistance to include multiple security rules such as: o Minimum length requirement Presence of at least one uppercase letter
#Presence of at least one lowercase letter
Presence of at least one digit
Presence of at least one special character
#2. Refactor the program 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 added rule 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, along with specific reasons if 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"
# Example usage
password_input = input("Enter password: ")
validation_result = validate_password(password_input)
print(validation_result)
```

OUTPUT:



OBSERVATION ;

The original program only checked the length of the password, so even a simple word like “abcdefgh” would be considered strong. The improved version adds multiple security rules, which makes the validation much more reliable. By checking for uppercase letters, lowercase letters, digits, and special characters, the password becomes harder to guess or break.

The use of a separate function validate_password() makes the code more organized and reusable. Meaningful variable names and proper structure improve readability. Adding a docstring and inline comments helps in understanding what the function does. Overall, the enhanced version is more secure, better structured, and follows good coding standards.