AI-Assisted Coding

Assignment-10.5

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

Name: K,sneha

Batch:20

HTNo.: 2303A51332

Task 1: Use AI to improve unclear variable names.

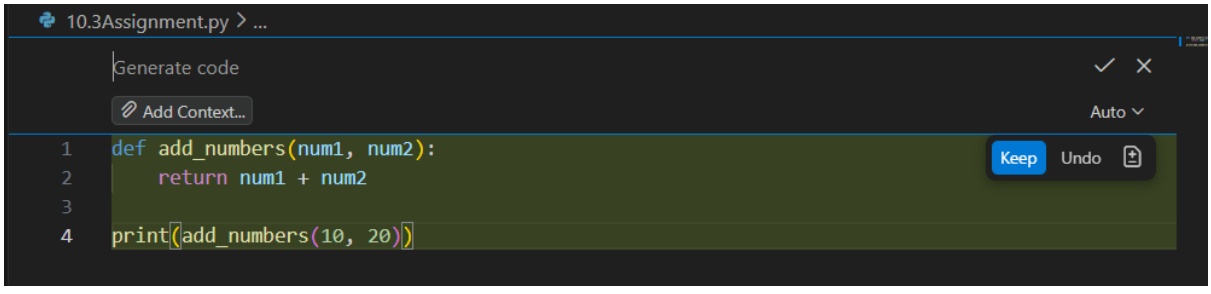
Prompt:

Improve the readability of the following Python code by replacing unclear function and variable names with meaningful and descriptive names. Follow Python naming conventions. Do not change the functionality of the code.

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

Code:

```
def add_numbers(num1, num2):
    return num1 + num2
print(add_numbers(10, 20))
```



Output:

```
30
```

Observation:

The original code uses unclear and non-descriptive names such as f, a, and b, which makes it difficult to understand the purpose of the function and variables. This reduces code readability and maintainability, especially in larger programs.After applying AI-based improvement, the function name was changed to add_numbers, and the variable names were changed to first_number and second_number. These meaningful names clearly describe the function's purpose and improve code clarity.

As a result, the rewritten code is easier to understand, more readable, and follows proper Python naming conventions while maintaining the same functionality.

Task 2: Use AI to add proper error handling.

Prompt:

Add proper error handling to the given Python code using try and except. Handle division by zero and invalid input type errors. Display clear and user-friendly error messages. Maintain the same functionality and use meaningful function and variable names.

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

Code:

```
def divide(a, b):
  try:
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
      raise TypeError("Both arguments must be numbers (int or float).")
    if b == 0:
      raise ValueError("Division by zero is not allowed. Divisor cannot be 0.")
    return a / b
  except TypeError as e:
    print(f"Type Error: {e}")
    return None
  except ValueError as e:
    print(f"Value Error: {e}")
    return None
```

except Exception as e:

print(f"An unexpected error occurred: {e}")

return None

print(divide(10, 2))

print(divide(10, 0))

print(divide("10", 5))

```python
#task2
def divide(a, b):
    try:
        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
            raise TypeError("Both arguments must be numbers (int or float).")
        if b == 0:
            raise ValueError("Division by zero is not allowed. Divisor cannot be 0.")
        return a / b
    except TypeError as e:
        print(f"Type Error: {e}")
        return None
    except ValueError as e:
        print(f"Value Error: {e}")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None

print(divide(10, 2))
print(divide(10, 0))
print(divide("10", 5))
```

Output:

```
5.0
Value Error: Division by zero is not allowed. Divisor cannot be 0.
None
Type Error: Both arguments must be numbers (int or float).
None
```

Observation:

The original code does not include error handling, so it crashes when a division by zero or invalid input occurs. After adding exception handling using try and except, the program can handle errors gracefully and display clear, user-friendly error messages instead of stopping abruptly. This improves the program's reliability, readability, and overall robustness while maintaining the same functionality.

Task3: Student Marks Processing System

Prompt:

Refactor the following Python code to improve readability, maintainability, and reliability. Follow PEP 8 coding standards. Replace unclear variable names with meaningful names, organize the logic into functions, and add appropriate comments and documentation (docstrings). Also include basic input validation to ensure the marks list is not empty, contains only numeric values, and marks are within a valid range. Add exception handling to display clear error messages without changing the program's main functionality.

```
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")
```

Code:

```
def calculate_grade(marks):
  """
  Calculate the average of marks and return the corresponding grade.

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

  Returns:
    str: The grade based on average marks (A, B, C, or F).

  Raises:
```

```
    ValueError: If marks list is empty, contains non-numeric values, or invalid ranges.
    TypeError: If marks is not a list.
  """
  try:
    if not isinstance(marks, list):
      raise TypeError("Marks must be provided as a list.")
    if len(marks) == 0:
      raise ValueError("Marks list cannot be empty.")
    for mark in marks:
      if not isinstance(mark, (int, float)):
        raise ValueError("All marks must be numeric values (int or float).")
      if not (0 <= mark <= 100):
        raise ValueError("All marks must be within the range 0-100.")
    average = sum(marks) / len(marks)
    if average >= 90:
      return "A"
    elif average >= 75:
      return "B"
    elif average >= 60:
      return "C"
    else:
      return "F"
  except TypeError as e:
    print(f"Type Error: {e}")
    return None
  except ValueError as e:
    print(f"Value Error: {e}")
```

return None

except Exception as e:

print(f"An unexpected error occurred: {e}")

return None

print(calculate_grade([78, 85, 90, 66, 88]))

print(calculate_grade([95, 98, 92]))

print(calculate_grade([]))

print(calculate_grade([78, "invalid", 90]))

print(calculate_grade([78, 105, 90]))

```python
def calculate_grade(marks):
    """
    Calculate the average of marks and return the corresponding grade.

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

    Returns:
        str: The grade based on average marks (A, B, C, or F).

    Raises:
        ValueError: If marks list is empty, contains non-numeric values, or invalid ranges.
        TypeError: If marks is not a list.
    """
    try:
        if not isinstance(marks, list):
            raise TypeError("Marks must be provided as a list.")

        if len(marks) == 0:
            raise ValueError("Marks list cannot be empty.")

        for mark in marks:
            if not isinstance(mark, (int, float)):
                raise ValueError("All marks must be numeric values (int or float).")
            if not (0 <= mark <= 100):
                raise ValueError("All marks must be within the range 0-100.")

        average = sum(marks) / len(marks)

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

```
        else:
            return "F"

    except TypeError as e:
        print(f"Type Error: {e}")
        return None
    except ValueError as e:
        print(f"Value Error: {e}")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None

print(calculate_grade([78, 85, 90, 66, 88]))
print(calculate_grade([95, 98, 92]))
print(calculate_grade([]))
print(calculate_grade([78, "invalid", 90]))
print(calculate_grade([78, 105, 90]))
```

Output:

```
B
A
Value Error: Marks list cannot be empty.
None
Value Error: All marks must be numeric values (int or float).
None
Value Error: All marks must be within the range 0-100.
None
```

Observation:

The original program had poor readability due to unclear variable names like t and a, lack of proper structure, and no comments or documentation. It also did not include input validation or error handling, which could cause the program to crash if invalid or empty data is provided.After refactoring using AI, the code now follows PEP 8 standards with meaningful variable and function names, proper indentation, and modular functions. Docstrings and comments improve understanding, while input validation ensures marks are numeric and within a valid range. Exception handling prevents crashes and displays clear error messages. Overall, the refactored code is more readable, maintainable, and reliable.

Task4: Use AI to add docstrings and inline comments to the following function.

Prompt:

Add proper docstrings and inline comments to the following Python function to improve readability and documentation. The docstring should explain the purpose of the function, its parameter, and its return value. Inline comments should explain the logic of the code. Do not change the functionality.

```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

Code:

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

    Parameters:
        n (int): The number whose factorial is to be calculated.
    Returns:
        int: Factorial of the given number.
    Raises:
        TypeError: If the input is not an integer.
        ValueError: If the input is a negative integer.
    """
    # Check if input is an integer
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")
    # Check if input is non-negative
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    # Initialize result to 1 (factorial base value)
    result = 1
    # Multiply numbers from 1 to n
    for i in range(1, n + 1):
        result *= i  # Multiply current result with i
```

```
    # Return the calculated factorial
    return result
# Example usage
print(factorial(5))  # Output: 120
print(factorial(0))  # Output: 1
```

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

    Parameters:
        n (int): The number whose factorial is to be calculated.

    Returns:
        int: Factorial of the given number.

    Raises:
        TypeError: If the input is not an integer.
        ValueError: If the input is a negative integer.
    """
    # Check if input is an integer
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")
    # Check if input is non-negative
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    # Initialize result to 1 (factorial base value)
    result = 1
    # Multiply numbers from 1 to n
    for i in range(1, n + 1):
        result *= i  # Multiply current result with i
    # Return the calculated factorial
    return result
# Example usage
print(factorial(5))  # Output: 120
print(factorial(0))  # Output: 1
```

Output:

```
120
1
```

Observation:

The original function had no documentation or comments, making it harder to understand its purpose and logic. After using AI to add docstrings and inline comments, the function becomes easier to read, understand, and maintain. Proper documentation helps other developers quickly understand the function's behavior, inputs, and outputs without analyzing the entire code.

Task5: Password Validation System (Enhanced)

Prompt:

Enhance the following Python password validation program to meet real-world security standards and improve code quality. Refactor the code to follow PEP 8 guidelines, use meaningful variable and function names, and organize the logic into reusable functions. Add validation rules including minimum length, at least one uppercase letter, one lowercase letter, one digit, and one special character. Include proper docstrings and inline comments explaining the logic. Also ensure the program provides clear feedback to the user about password strength without changing the core purpose of validating a password.

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

Code:

```
def validate_password(password):
  """
  Validate password against security standards.

  Args:
    password (str): The password to validate.

  Returns:
    tuple: (is_valid, feedback) where is_valid is bool and feedback is str.
```

```
    Raises:
        TypeError: If password is not a string.
    """
    if not isinstance(password, str):
        raise TypeError("Password must be a string.")
    feedback = []
    # Check minimum length (8 characters)
    if len(password) < 8:
        feedback.append("Password must be at least 8 characters long.")
    # Check for at least one uppercase letter
    if not re.search(r'[A-Z]', password):
        feedback.append("Password must contain at least one uppercase letter.")
    # Check for at least one lowercase letter
    if not re.search(r'[a-z]', password):
        feedback.append("Password must contain at least one lowercase letter.")
    # Check for at least one digit
    if not re.search(r'\d', password):
        feedback.append("Password must contain at least one digit.")
    # Check for at least one special character
    if not re.search(r'[!@#$%^&*()_+\-=\[\]{};:\'",.<>?/\\|`~]', password):
        feedback.append("Password must contain at least one special character.")
    # Determine if password is valid
    is_valid = len(feedback) == 0
    return is_valid, feedback
def display_password_strength(is_valid, feedback):
    """
    Display password validation result and feedback.
```

```
    Args:
        is_valid (bool): Whether password is valid.
        feedback (list): List of validation messages.
    """
    if is_valid:
        print("Strong")
    else:
        print("Weak")
        for message in feedback:
            print(f"  - {message}")
# Main program
pwd = input("Enter password: ")
try:
    is_valid, feedback = validate_password(pwd)
    display_password_strength(is_valid, feedback)
except TypeError as e:
    print(f"Type Error: {e}")
```

```python
def validate_password(password):
    """
    Validate password against security standards.

    Args:
        password (str): The password to validate.

    Returns:
        tuple: (is_valid, feedback) where is_valid is bool and feedback is str.

    Raises:
        TypeError: If password is not a string.
    """
    if not isinstance(password, str):
        raise TypeError("Password must be a string.")
    feedback = []
    # Check minimum length (8 characters)
    if len(password) < 8:
        feedback.append("Password must be at least 8 characters long.")
    # Check for at least one uppercase letter
    if not re.search(r'[A-Z]', password):
        feedback.append("Password must contain at least one uppercase letter.")
    # Check for at least one lowercase letter
    if not re.search(r'[a-z]', password):
        feedback.append("Password must contain at least one lowercase letter.")
    # Check for at least one digit
    if not re.search(r'\d', password):
        feedback.append("Password must contain at least one digit.")
    # Check for at least one special character
    if not re.search(r'[!@#$%^&*()_+\-=\[\]{};:\'",.<>?/\\|`~]', password):
        feedback.append("Password must contain at least one special character.")
    # Determine if password is valid
```

```
    is_valid = len(feedback) == 0
    return is_valid, feedback
def display_password_strength(is_valid, feedback):
    """
    Display password validation result and feedback.

    Args:
        is_valid (bool): Whether password is valid.
        feedback (list): List of validation messages.
    """
    if is_valid:
        print("Strong")
    else:
        print("Weak")
        for message in feedback:
            print(f"  - {message}")
# Main program
pwd = input("Enter password: ")
try:
    is_valid, feedback = validate_password(pwd)
    display_password_strength(is_valid, feedback)
except TypeError as e:
    print(f"Type Error: {e}")
```

```
Enter password: Abcdef@9989
Strong
```

Observation:

The original password validation program only checks for a minimum length of 8 characters, which is not sufficient for real-world security. It allows weak passwords that are easy to guess or crack. The code also lacks proper structure, meaningful variable names, functions, comments, and documentation, making it less readable and harder to maintain.After using AI to enhance the program, multiple security rules were added, including checks for uppercase letters, lowercase letters, digits, and special characters. The code was refactored to follow PEP 8 standards, with meaningful function and variable names, proper docstrings, and inline comments. The validation logic was organized into reusable functions, improving modularity and maintainability.