ASSIGNMENT-10.5

Name : M.Akshay patel Hall : 2303A51745 Batch : 11

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.

CODE:

```
[15]
✓ 0s
# Expected Output (rewritten code after applying AI suggestions)
def add_numbers(num1:int, num2:int) -> int:
    """This function adds two numbers together."""
    return num1 + num2

result = add_numbers(10, 20)
print(f"The sum of 10 and 20 is: {result}")
```

OUTPUT:

```
print(f"The sum of 10 and 20 is: {result}")

...  The sum of 10 and 20 is: 30
```

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

CODE:

```
[20]
✓ 0s
def divide(a: int | float, b: int | float) -> int | float | str:
    """
    Divides two numbers and handles a potential ZeroDivisionError.

    Args:
        a (int | float): The numerator.
        b (int | float): The denominator.

    Returns:
        int | float | str: The result of the division, or an error message
                           if division by zero is attempted.
    """
    try:
        # Attempt to perform the division
        result = a / b
        return result
    except ZeroDivisionError:
        # Catch the specific error when attempting to divide by zero
        return "Error: Cannot divide by zero!"

# Example usage with valid input
print(f"Division of 10 by 2: {divide(10, 2)}")

# Example usage with potential error (division by zero)
print(f"Division of 10 by 0: {divide(10, 0)}")
```

OUTPUT:

```
Division of 10 by 2: 5.0
Division of 10 by 0: Error: Cannot divide by zero!
```

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")
```

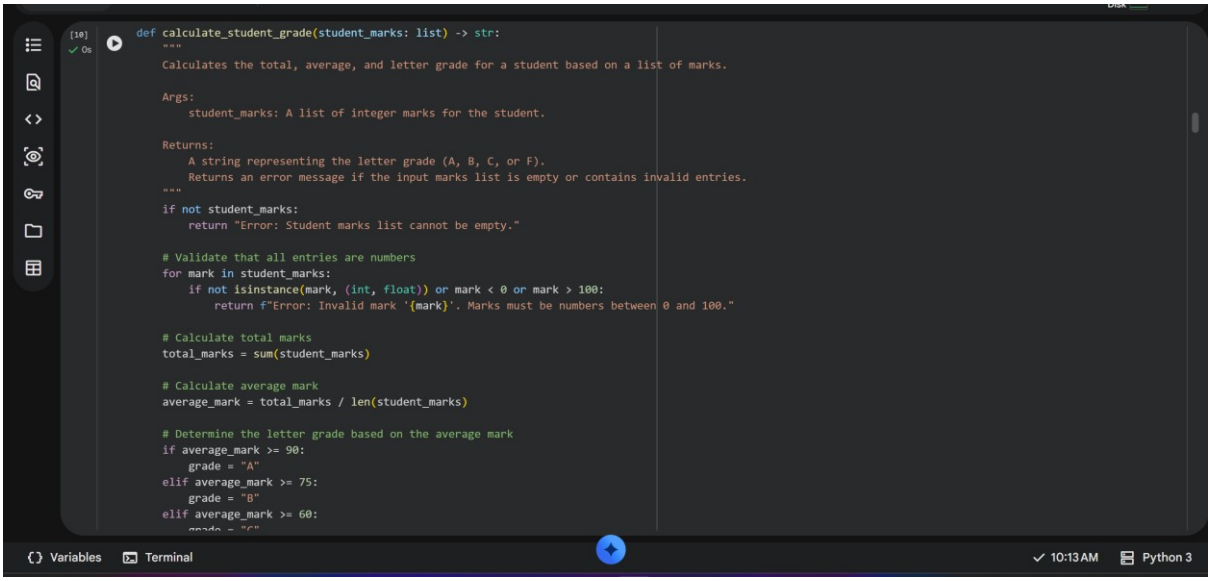
Task:

• Use AI to refactor the code to follow PEP 8 standards.

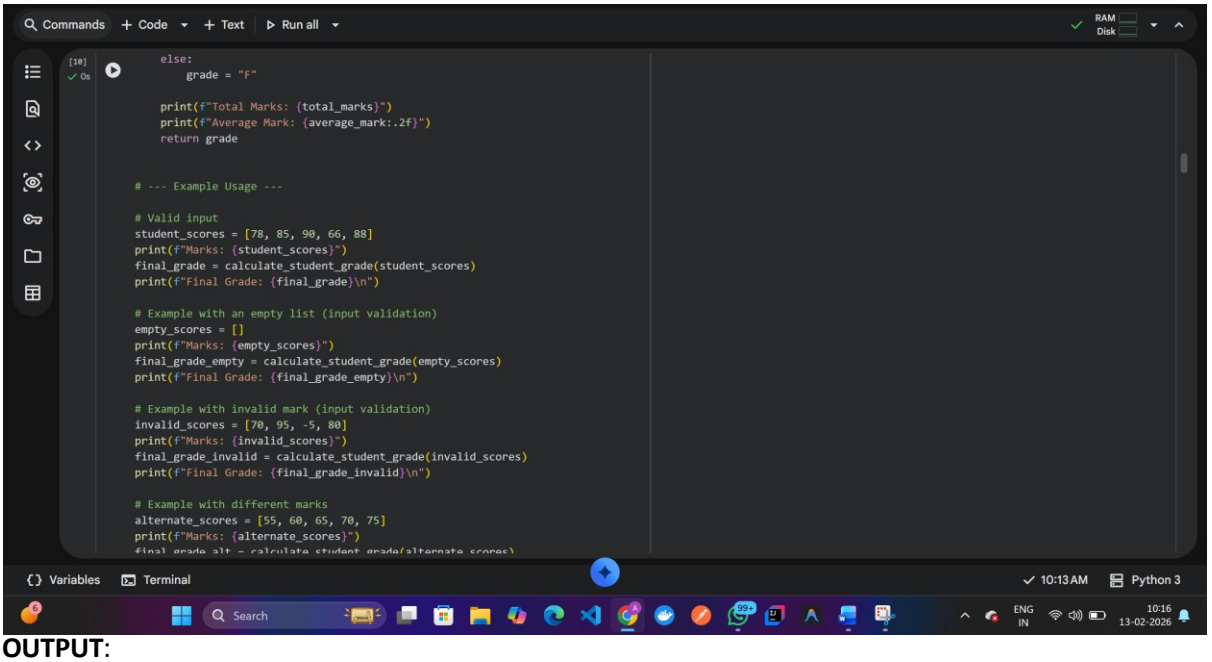
• Add meaningful variable names, functions, and comments.

• Add basic input validation and documentation.

CODE:



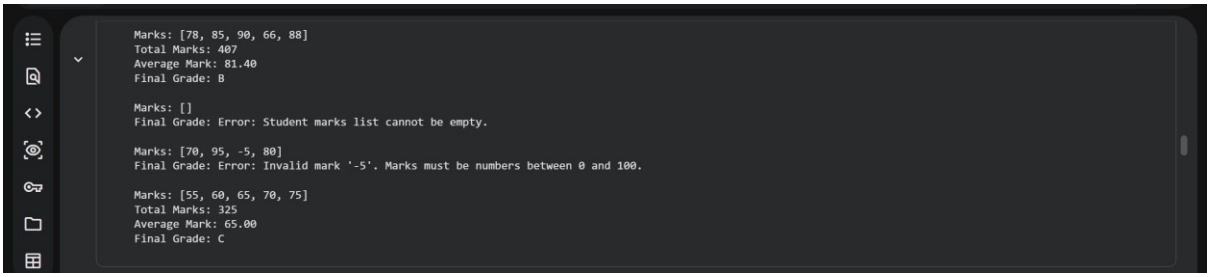
```
def calculate_student_grade(student_marks: list) -> str:
    """
    Calculates the total, average, and letter grade for a student based on a list of marks.

    Args:
        student_marks: A list of integer marks for the student.

    Returns:
        A string representing the letter grade (A, B, C, or F).
        Returns an error message if the input marks list is empty or contains invalid entries.
    """
    if not student_marks:
        return "Error: Student marks list cannot be empty."

    # Validate that all entries are numbers
    for mark in student_marks:
        if not isinstance(mark, (int, float)) or mark < 0 or mark > 100:
            return f"Error: Invalid mark '{mark}'. Marks must be numbers between 0 and 100."

    # Calculate total marks
    total_marks = sum(student_marks)

    # Calculate average mark
    average_mark = total_marks / len(student_marks)

    # Determine the letter grade based on the average mark
    if average_mark >= 90:
        grade = "A"
    elif average_mark >= 75:
        grade = "B"
    elif average_mark >= 60:
        grade = "C"
```



```
    else:
        grade = "F"

    print(f"Total Marks: {total_marks}")
    print(f"Average Mark: {average_mark:.2f}")
    return grade


# --- Example Usage ---

# Valid input
student_scores = [78, 85, 90, 66, 88]
print(f"Marks: {student_scores}")
final_grade = calculate_student_grade(student_scores)
print(f"Final Grade: {final_grade}\n")

# Example with an empty list (input validation)
empty_scores = []
print(f"Marks: {empty_scores}")
final_grade_empty = calculate_student_grade(empty_scores)
print(f"Final Grade: {final_grade_empty}\n")

# Example with invalid mark (input validation)
invalid_scores = [70, 95, -5, 80]
print(f"Marks: {invalid_scores}")
final_grade_invalid = calculate_student_grade(invalid_scores)
print(f"Final Grade: {final_grade_invalid}\n")

# Example with different marks
alternate_scores = [55, 60, 65, 70, 75]
print(f"Marks: {alternate_scores}")
```

OUTPUT:

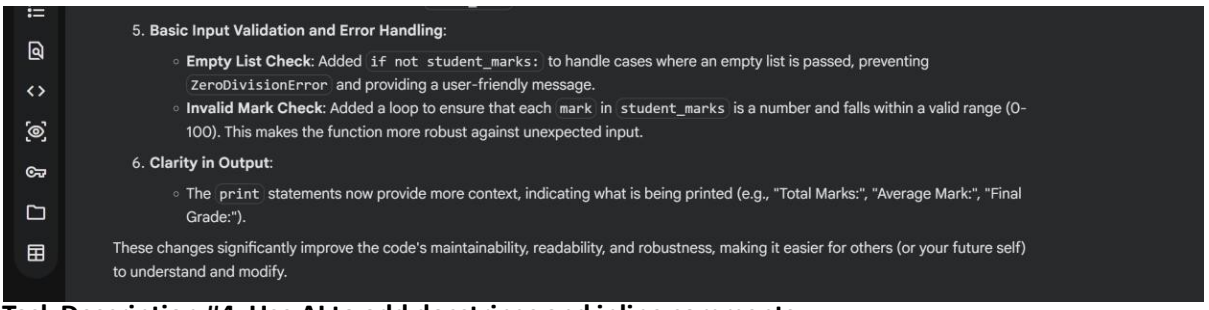
EXPLANATION:



```
Marks: [78, 85, 90, 66, 88]
Total Marks: 407
Average Mark: 81.40
Final Grade: B

Marks: []
Final Grade: Error: Student marks list cannot be empty.

Marks: [70, 95, -5, 80]
Final Grade: Error: Invalid mark '-5'. Marks must be numbers between 0 and 100.

Marks: [55, 60, 65, 70, 75]
Total Marks: 325
Average Mark: 65.00
Final Grade: C
```

Explanation of Refactoring Changes:

1. **Meaningful Names (PEP 8)**:
   - `marks` -> `student_marks` : More descriptive of the data.
   - `t` -> `total_marks` : Clearly indicates what the variable holds.
   - `i` -> `mark` : Better name for individual items in the loop.
   - `a` -> `average_mark` : Explicitly states the calculated value.
   - `f, a, b, c` -> `calculate_student_grade` : A function name that explains its purpose.
2. **Function Encapsulation**:
   - The logic is now contained within a function `calculate_student_grade`. This improves modularity, reusability, and testability. It also makes the code easier to understand by breaking it into logical units.
3. **Comments and Docstrings (PEP 257 & PEP 8)**:
   - A docstring has been added to the `calculate_student_grade` function, explaining its purpose, arguments, and return value. This is crucial for documentation and helps anyone reading the code understand its functionality without needing to delve into the implementation details.
   - Inline comments are used to clarify specific steps, such as calculating total marks or determining the grade.
4. **PEP 8 Styling**:
   - Consistent indentation (4 spaces) and spacing around operators (`=`, `+`, `/`) are applied.
   - Blank lines are used to separate logical blocks within the function for improved readability.
   - Function and variable names follow `snake_case` conventions.



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

to the following function.

def factorial(n):

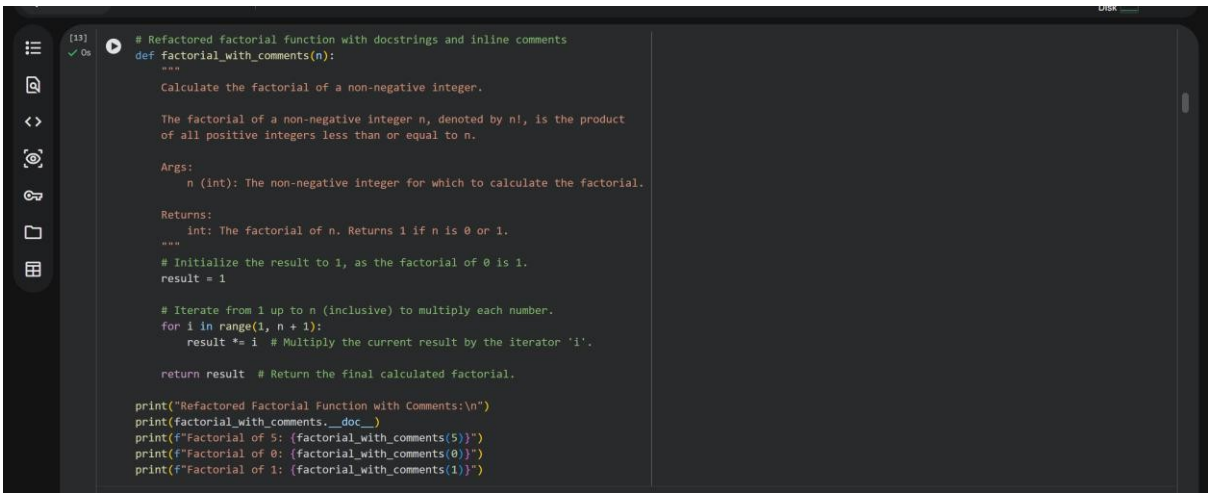
result = 1

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

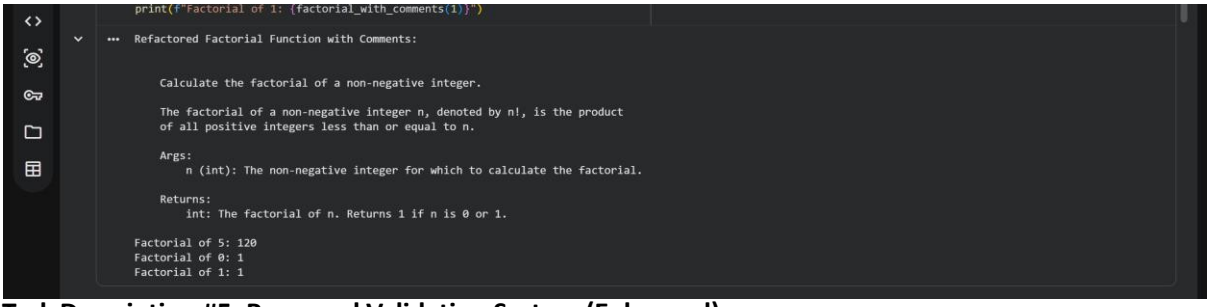
result *= i

return result

CODE:



OUTPUT:



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:

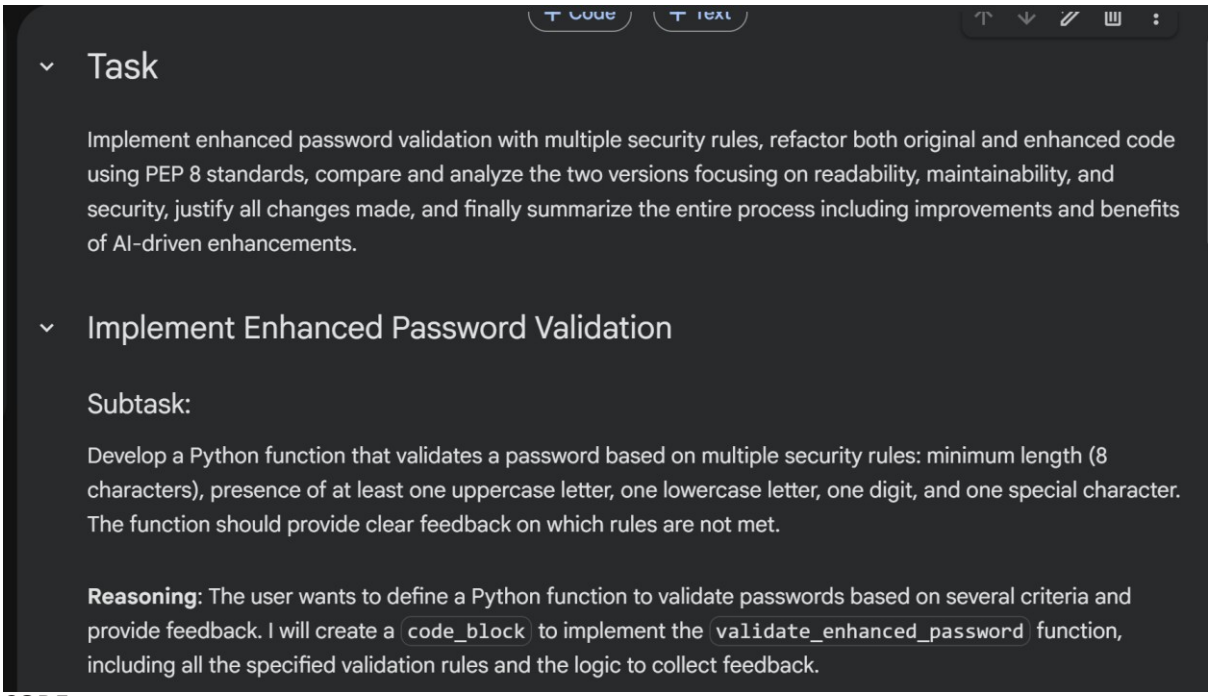
o Code readability and structure

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.

TASK EXPLANATION:



CODE:

```
def validate_enhanced_password(password):
    feedback = []
    is_valid = True

    # Rule 1: Minimum length of 8 characters
    if len(password) < 8:
        feedback.append("Password must be at least 8 characters long.")
        is_valid = False

    # Rule 2: At least one uppercase letter
    if not any(c.isupper() for c in password):
        feedback.append("Password must contain at least one uppercase letter.")
        is_valid = False

    # Rule 3: At least one lowercase letter
    if not any(c.islower() for c in password):
        feedback.append("Password must contain at least one lowercase letter.")
        is_valid = False

    # Rule 4: At least one digit
    if not any(c.isdigit() for c in password):
        feedback.append("Password must contain at least one digit.")
        is_valid = False

    # Rule 5: At least one special character
    special_characters = "!@#$%^&*()-_+=[]{}|;:'\"<>,./?`~"
    if not any(c in special_characters for c in password):
        feedback.append("Password must contain at least one special character (e.g., !@#$%^&*()).")
        is_valid = False

    return is_valid, feedback
```

```
The `validate_enhanced_password` function has been defined.
```

Refactor Programs

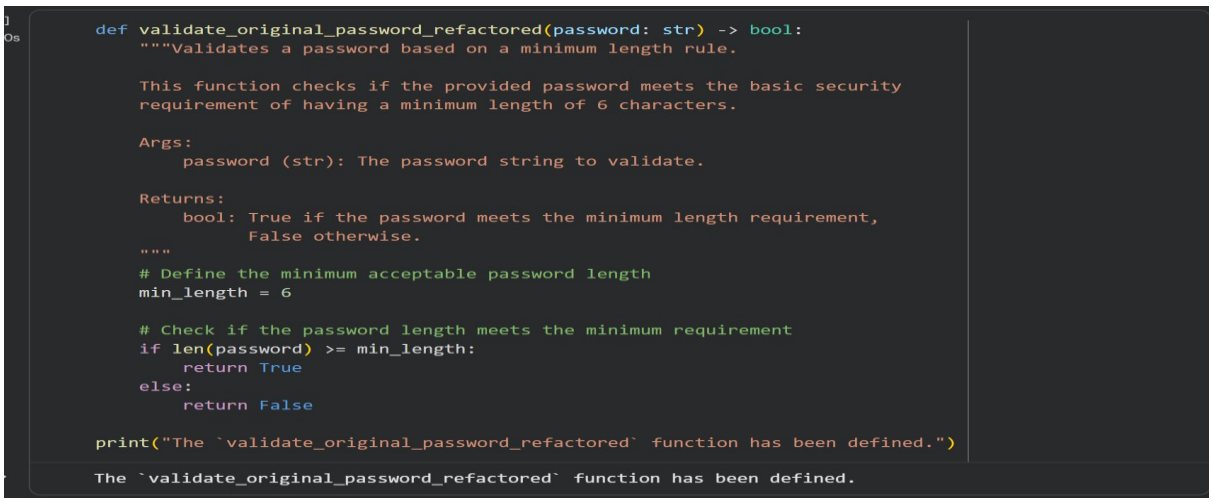
Subtask:

Implement a simple 'original' password validation function for later refactoring and comparison.

Reasoning: I will implement a simple password validation function named `validate_original_password` as requested, which checks if the password length is at least 6 characters and returns a boolean.

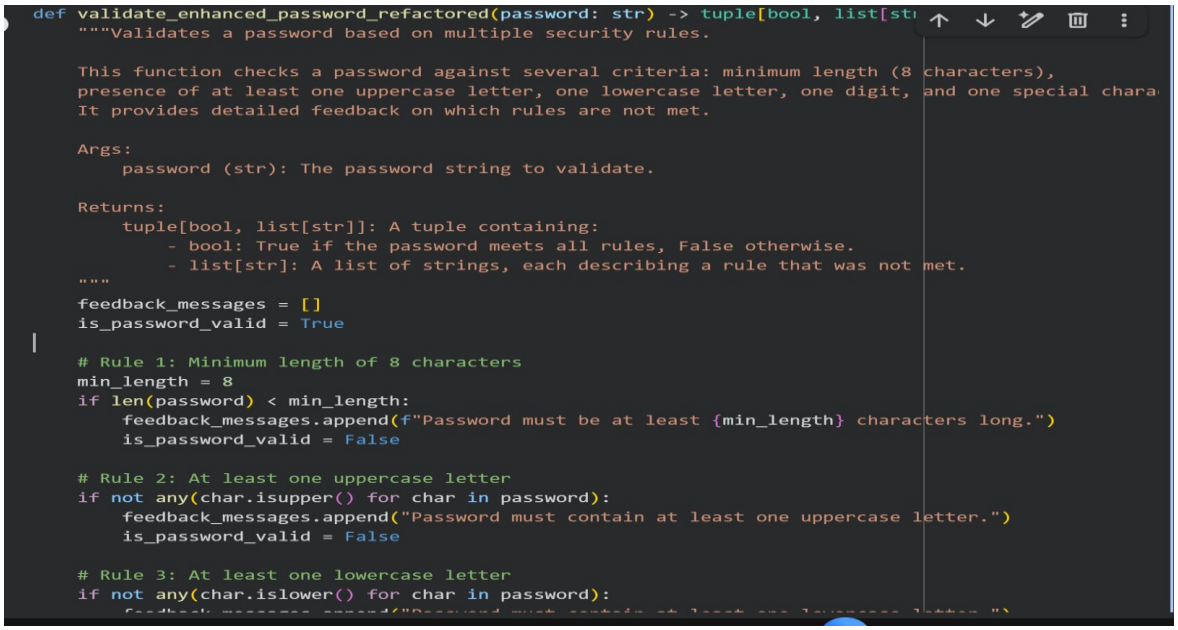
```
[2]
✓ 0s
def validate_original_password(password):
    # Rule: Minimum length of 6 characters
    if len(password) >= 6:
        return True
    else:
        return False

print("The `validate_original_password` function has been defined.")
```

```
... The `validate_original_password` function has been defined.
```

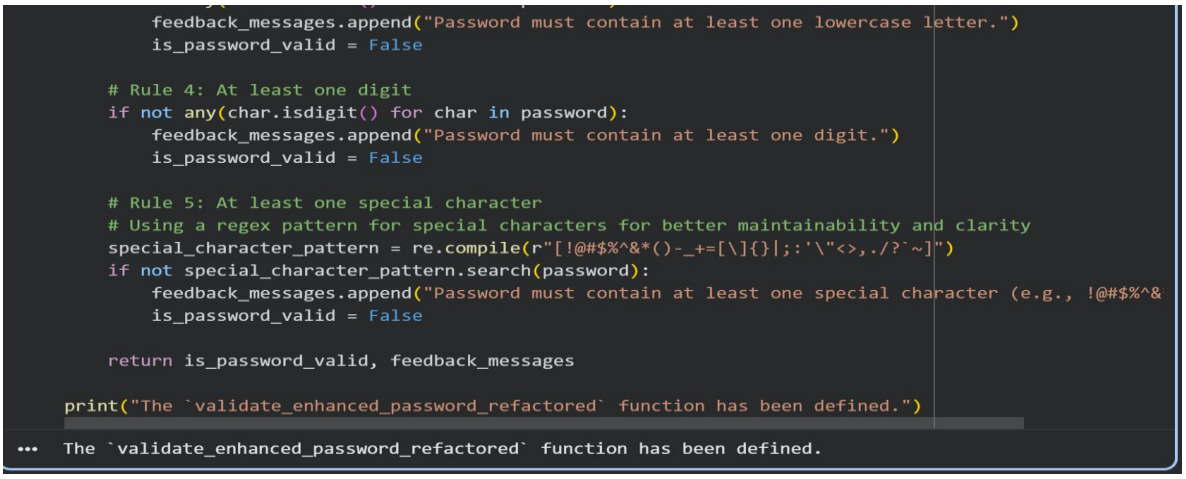


```
0s
def validate_original_password_refactored(password: str) -> bool:
    """Validates a password based on a minimum length rule.

    This function checks if the provided password meets the basic security
    requirement of having a minimum length of 6 characters.

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

    Returns:
        bool: True if the password meets the minimum length requirement,
              False otherwise.
    """
    # Define the minimum acceptable password length
    min_length = 6

    # Check if the password length meets the minimum requirement
    if len(password) >= min_length:
        return True
    else:
        return False

print("The `validate_original_password_refactored` function has been defined.")
```

```
The `validate_original_password_refactored` function has been defined.
```



```
def validate_enhanced_password_refactored(password: str) -> tuple[bool, list[str]]:
    """Validates a password based on multiple security rules.

    This function checks a password against several criteria: minimum length (8 characters),
    presence of at least one uppercase letter, one lowercase letter, one digit, and one special chara
    It provides detailed feedback on which rules are not met.

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

    Returns:
        tuple[bool, list[str]]: A tuple containing:
            - bool: True if the password meets all rules, False otherwise.
            - list[str]: A list of strings, each describing a rule that was not met.
    """
    feedback_messages = []
    is_password_valid = True

    # Rule 1: Minimum length of 8 characters
    min_length = 8
    if len(password) < min_length:
        feedback_messages.append(f"Password must be at least {min_length} characters long.")
        is_password_valid = False

    # Rule 2: At least one uppercase letter
    if not any(char.isupper() for char in password):
        feedback_messages.append("Password must contain at least one uppercase letter.")
        is_password_valid = False

    # Rule 3: At least one lowercase letter
    if not any(char.islower() for char in password):
```



```
            feedback_messages.append("Password must contain at least one lowercase letter.")
            is_password_valid = False

        # Rule 4: At least one digit
        if not any(char.isdigit() for char in password):
            feedback_messages.append("Password must contain at least one digit.")
            is_password_valid = False

        # Rule 5: At least one special character
        # Using a regex pattern for special characters for better maintainability and clarity
        special_character_pattern = re.compile(r"[!@#$%^&*()-_+=[\]{}|;:'\"<>,./?`~]")
        if not special_character_pattern.search(password):
            feedback_messages.append("Password must contain at least one special character (e.g., !@#$%^&
            is_password_valid = False

        return is_password_valid, feedback_messages

    print("The `validate_enhanced_password_refactored` function has been defined.")
```

```
...  The `validate_enhanced_password_refactored` function has been defined.
```

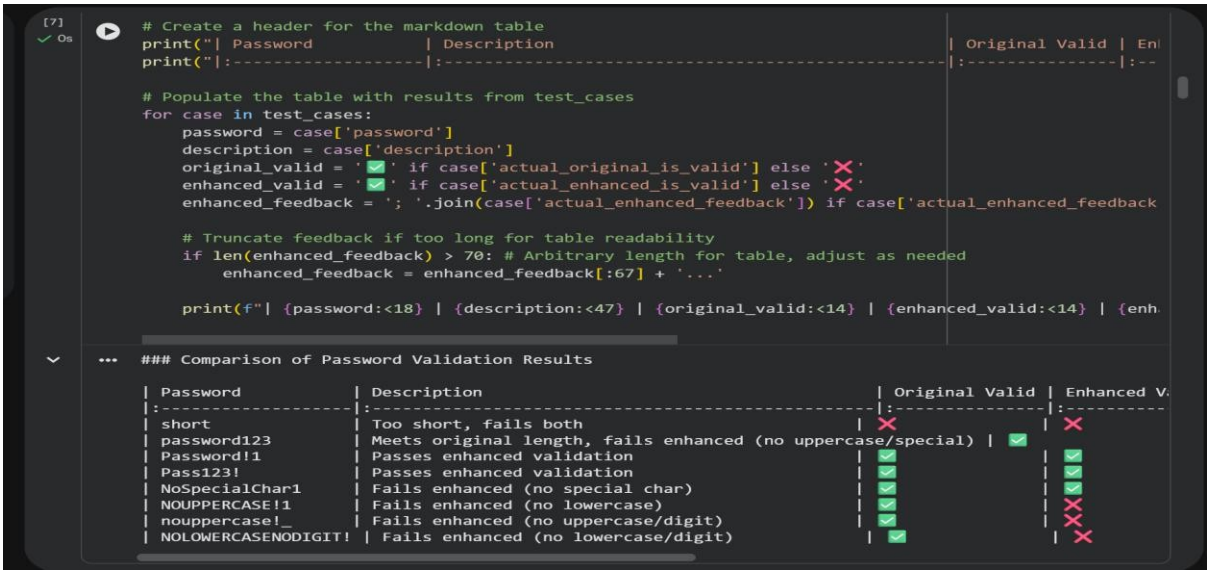
TEST CASES:

```
test_cases = [
  {
    "password": "short",
    "description": "Too short, fails both",
    "expected_original": False,
    "expected_enhanced_is_valid": False,
    "expected_enhanced_feedback_contains": [
      "Password must be at least 8 characters long.",
      "Password must contain at least one uppercase letter.",
      "Password must contain at least one digit.",
      "Password must contain at least one special character."
    ]
  },
  {
    "password": "password123",
    "description": "Meets original length, fails enhanced (no uppercase/special)",
    "expected_original": True,
    "expected_enhanced_is_valid": False,
    "expected_enhanced_feedback_contains": [
      "Password must contain at least one uppercase letter.",
      "Password must contain at least one special character."
    ]
```

```
},
{
  "password": "Password!1",
  "description": "Passes enhanced validation",
  "expected_original": True,
  "expected_enhanced_is_valid": True,
  "expected_enhanced_feedback_contains": []
},
{
  "password": "Pass123!",
  "description": "Passes enhanced validation",
  "expected_original": True,
  "expected_enhanced_is_valid": True,
  "expected_enhanced_feedback_contains": []
},
{
  "password": "NoSpecialChar1",
  "description": "Fails enhanced (no special char)",
  "expected_original": True,
  "expected_enhanced_is_valid": False,
  "expected_enhanced_feedback_contains": [
    "Password must contain at least one special character."
  ]
},
{
  "password": "NOUPPERCASE!1",
  "description": "Fails enhanced (no lowercase)",
  "expected_original": True,
  "expected_enhanced_is_valid": False,
  "expected_enhanced_feedback_contains": [
    "Password must contain at least one lowercase letter."
```

```
        ]
    },
    {
        "password": "nouppercase!_",
        "description": "Fails enhanced (no uppercase/digit)",
        "expected_original": True,
        "expected_enhanced_is_valid": False,
        "expected_enhanced_feedback_contains": [
            "Password must contain at least one uppercase letter.",
            "Password must contain at least one digit."
        ]
    },
    {
        "password": "NOLOWERCASENODIGIT!",
        "description": "Fails enhanced (no lowercase/digit)",
        "expected_original": True,
        "expected_enhanced_is_valid": False,
        "expected_enhanced_feedback_contains": [
            "Password must contain at least one lowercase letter.",
            "Password must contain at least one digit."
        ]
    }
]
print(f"Defined {len(test_cases)} password test cases.")
```

OUTPUT:



```
[7]
✓ 0s

# Create a header for the markdown table
print("| Password           | Description                                                | Original Valid | En
print("|:-------------------|:-----------------------------------------------------------|:---------------|:--

# Populate the table with results from test_cases
for case in test_cases:
    password = case['password']
    description = case['description']
    original_valid = '✅' if case['actual_original_is_valid'] else '❌'
    enhanced_valid = '✅' if case['actual_enhanced_is_valid'] else '❌'
    enhanced_feedback = '; '.join(case['actual_enhanced_feedback']) if case['actual_enhanced_feedback

    # Truncate feedback if too long for table readability
    if len(enhanced_feedback) > 70: # Arbitrary length for table, adjust as needed
        enhanced_feedback = enhanced_feedback[:67] + '...'

    print(f"| {password:<18} | {description:<47} | {original_valid:<14} | {enhanced_valid:<14} | {enh
```

```
### Comparison of Password Validation Results

| Password           | Description                                                | Original Valid | Enhanced V
|:-------------------|:-----------------------------------------------------------|:---------------|:----------
| short              | Too short, fails both                                      | ❌             | ❌
| password123        | Meets original length, fails enhanced (no uppercase/special) | ✅          |
| Password!1         | Passes enhanced validation                                 | ✅             | ✅
| Pass123!           | Passes enhanced validation                                 | ✅             | ✅
| NoSpecialChar1     | Fails enhanced (no special char)                           | ✅             | ✅
| NOUPPERCASE!1      | Fails enhanced (no lowercase)                              | ✅             | ❌
| nouppercase!_      | Fails enhanced (no uppercase/digit)                        | ✅             | ❌
| NOLOWERCASENODIGIT! | Fails enhanced (no lowercase/digit)                       | ✅             | ❌
```