AI ASSISTED CODING

Sai Thrishool **2303A51127**

BATCH – 03 **13 – 02 – 2026**

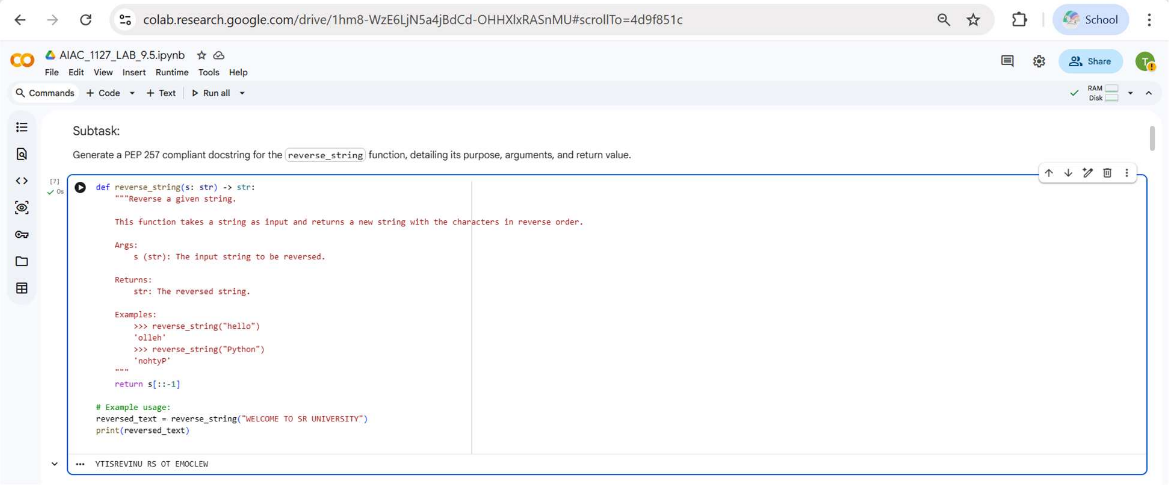
ASSIGNMENT – 9.5

Lab 9.5: Documentation Generation -Automatic documentation and code comments

Task1: String Utilities Function

Prompt 1: Generate a PEP 257 compliant docstring for the reverse_string function, detailing its purpose, arguments, and return value.

CODE & OUTPUT:



colab.research.google.com/drive/1hm8-WzE6LjN5a4jBdCd-OHHXlxRASnMU#scrollTo=4d9f851c
AIAC_1127_LAB_9.5.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
Share
School

Subtask:

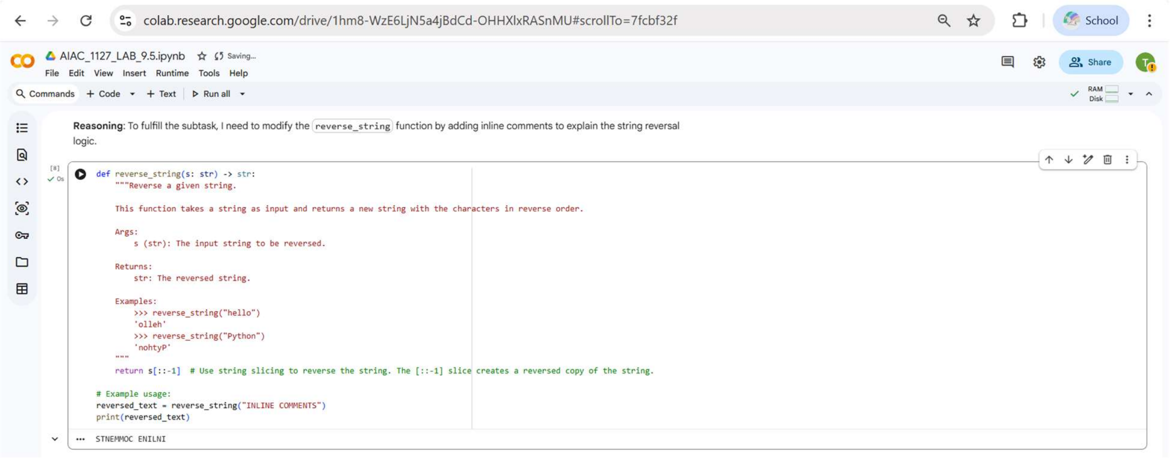
Generate a PEP 257 compliant docstring for the `reverse_string` function, detailing its purpose, arguments, and return value.

```
def reverse_string(s: str) -> str:
    """Reverse a given string.

    This function takes a string as input and returns a new string with the characters in reverse order.

    Args:
        s (str): The input string to be reversed.

    Returns:
        str: The reversed string.

    Examples:
        >>> reverse_string("hello")
        'olleh'
        >>> reverse_string("Python")
        'nohtyP'
    """
    return s[::-1]

# Example usage:
reversed_text = reverse_string("WELCOME TO SR UNIVERSITY")
print(reversed_text)
```

```
YTISREVINU RS OT EMOCLEW
```

Prompt 2: Add appropriate inline comments to the reverse_string function to explain each line or logical block of code.

CODE & OUTPUT:



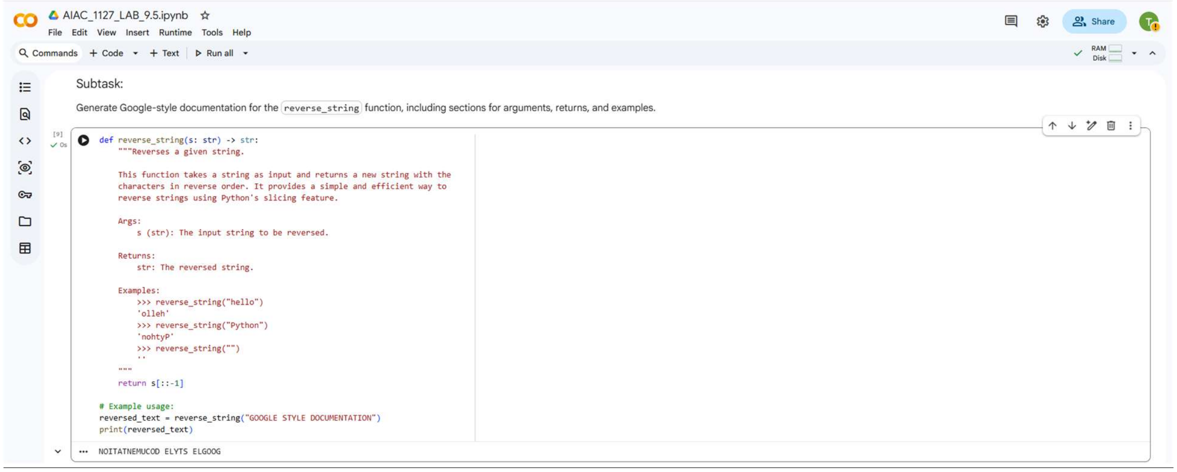
Reasoning: To fulfill the subtask, I need to modify the `reverse_string` function by adding inline comments to explain the string reversal logic.

```
def reverse_string(s: str) -> str:
    """Reverse a given string.

    This function takes a string as input and returns a new string with the characters in reverse order.

    Args:
        s (str): The input string to be reversed.

    Returns:
        str: The reversed string.

    Examples:
        >>> reverse_string("hello")
        'olleh'
        >>> reverse_string("Python")
        'nohtyP'
    """
    return s[::-1]  # Use string slicing to reverse the string. The [::-1] slice creates a reversed copy of the string.

# Example usage:
reversed_text = reverse_string("INLINE COMMENTS")
print(reversed_text)
```

```
STNEMMOC ENILNI
```

Prompt 3: Generate Google-style documentation for the reverse_string function, including sections for arguments, returns, and examples.

CODE & OUTPUT:



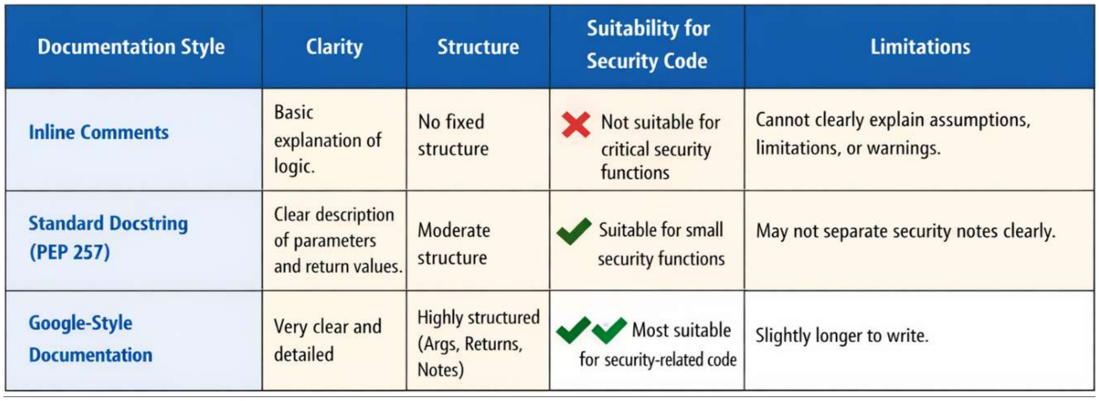
Subtask:

Generate Google-style documentation for the `reverse_string` function, including sections for arguments, returns, and examples.

```
def reverse_string(s: str) -> str:
    """Reverses a given string.

    This function takes a string as input and returns a new string with the
    characters in reverse order. It provides a simple and efficient way to
    reverse strings using Python's slicing feature.

    Args:
        s (str): The input string to be reversed.

    Returns:
        str: The reversed string.

    Examples:
        >>> reverse_string("hello")
        'olleh'
        >>> reverse_string("Python")
        'nohtyP'
        >>> reverse_string("")
        ''
    """
    return s[::-1]

# Example usage:
reversed_text = reverse_string("GOOGLE STYLE DOCUMENTATION")
print(reversed_text)
```

```
NOITATNEMUCOD ELYTS ELGOOG
```

Comparison:



Documentation Style	Clarity	Structure	Suitability for Security Code	Limitations
Inline Comments	Basic explanation of logic.	No fixed structure	✗ Not suitable for critical security functions	Cannot clearly explain assumptions, limitations, or warnings.
Standard Docstring (PEP 257)	Clear description of parameters and return values.	Moderate structure	✓ Suitable for small security functions	May not separate security notes clearly.
Google-Style Documentation	Very clear and detailed	Highly structured (Args, Returns, Notes)	✓✓ Most suitable for security-related code	Slightly longer to write.

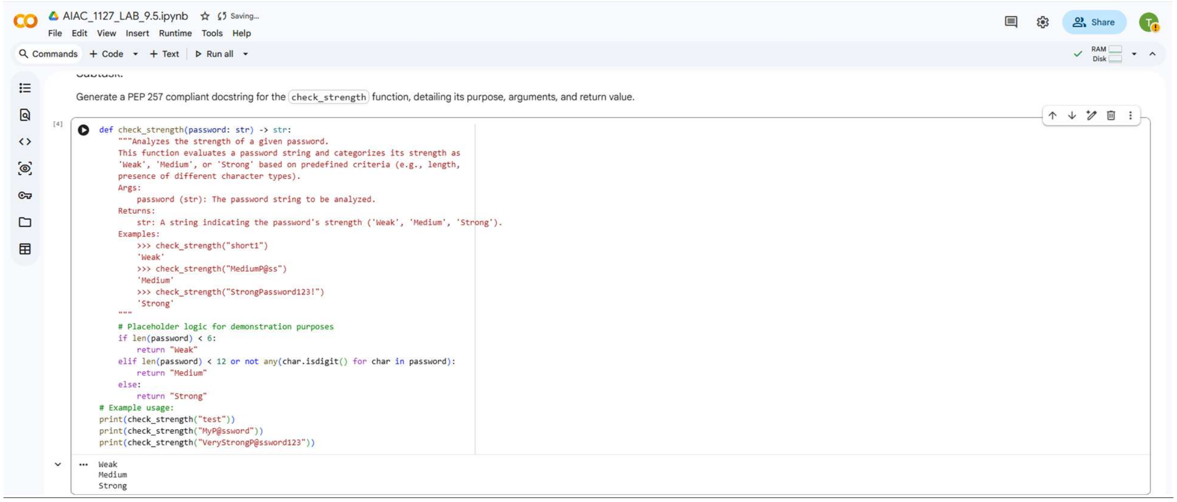
JUSTIFICATION FOR BEST DOCUMENT STYLE:

Google-style documentation is the most appropriate for security-related code because it provides a clear and structured format using sections like Args, Returns, and Notes.

TASK 2: Password Strength Checker

Prompt 1: Generate a PEP 257 compliant docstring for the check_strength function, detailing its purpose, arguments, and return value.

CODE & OUTPUT:



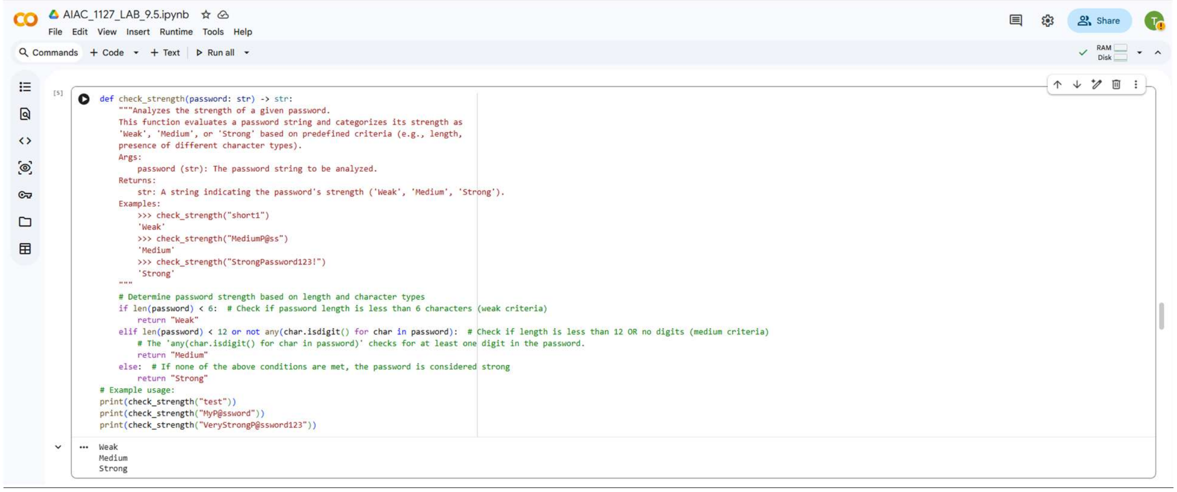
Generate a PEP 257 compliant docstring for the `check_strength` function, detailing its purpose, arguments, and return value.

```
[4]
def check_strength(password: str) -> str:
    """Analyzes the strength of a given password.
    This function evaluates a password string and categorizes its strength as
    'Weak', 'Medium', or 'Strong' based on predefined criteria (e.g., length,
    presence of different character types).
    Args:
        password (str): The password string to be analyzed.
    Returns:
        str: A string indicating the password's strength ('Weak', 'Medium', 'Strong').
    Examples:
        >>> check_strength("short1")
        'Weak'
        >>> check_strength("MediumP@ss")
        'Medium'
        >>> check_strength("StrongPassword123!")
        'Strong'
    """
    # Placeholder logic for demonstration purposes
    if len(password) < 6:
        return "Weak"
    elif len(password) < 12 or not any(char.isdigit() for char in password):
        return "Medium"
    else:
        return "Strong"
# Example usage:
print(check_strength("test"))
print(check_strength("MyP@ssword"))
print(check_strength("VeryStrongP@ssword123"))
```

```
Weak
Medium
Strong
```

Prompt 2: Add appropriate inline comments to the check_strength function to explain each line or logical block of code.

CODE & OUTPUT:

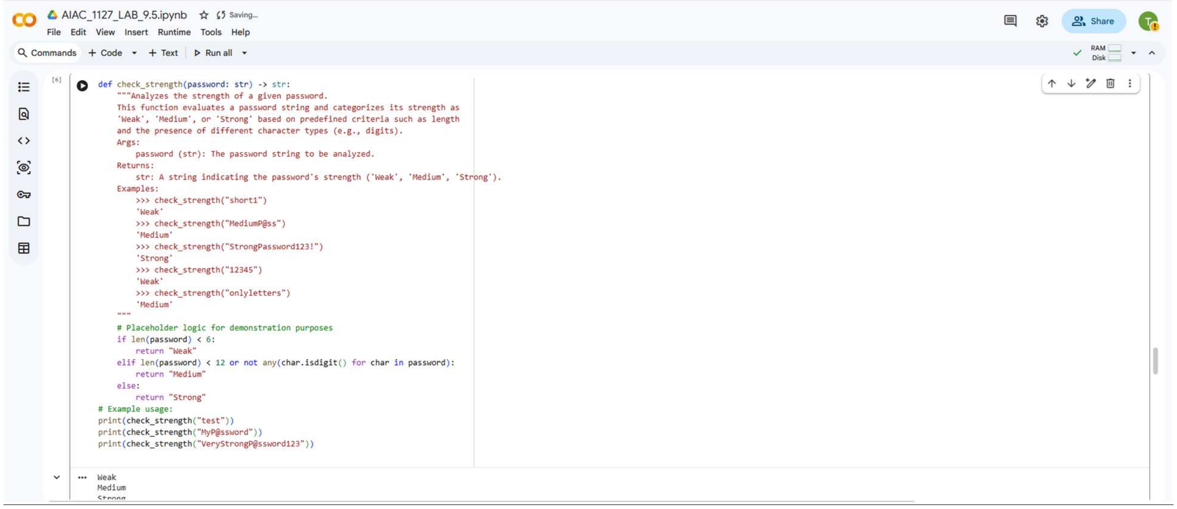


```
def check_strength(password: str) -> str:
    """Analyzes the strength of a given password.
    This function evaluates a password string and categorizes its strength as
    'Weak', 'Medium', or 'Strong' based on predefined criteria (e.g., length,
    presence of different character types).
    Args:
        password (str): The password string to be analyzed.
    Returns:
        str: A string indicating the password's strength ('Weak', 'Medium', 'Strong').
    Examples:
        >>> check_strength("short1")
        'Weak'
        >>> check_strength("MediumP@ss")
        'Medium'
        >>> check_strength("StrongPassword123!")
        'Strong'
    """
    # Determine password strength based on length and character types
    if len(password) < 6:  # Check if password length is less than 6 characters (weak criteria)
        return "Weak"
    elif len(password) < 12 or not any(char.isdigit() for char in password):  # Check if length is less than 12 OR no digits (medium criteria)
        # The 'any(char.isdigit() for char in password)' checks for at least one digit in the password.
        return "Medium"
    else:  # If none of the above conditions are met, the password is considered strong
        return "Strong"
# Example usage:
print(check_strength("test"))
print(check_strength("MyP@ssword"))
print(check_strength("VeryStrongP@ssword123"))
```

```
Weak
Medium
Strong
```

Prompt 3: Generate Google-style documentation for the check_strength function, including sections for arguments, returns, and examples.

CODE & OUTPUT:



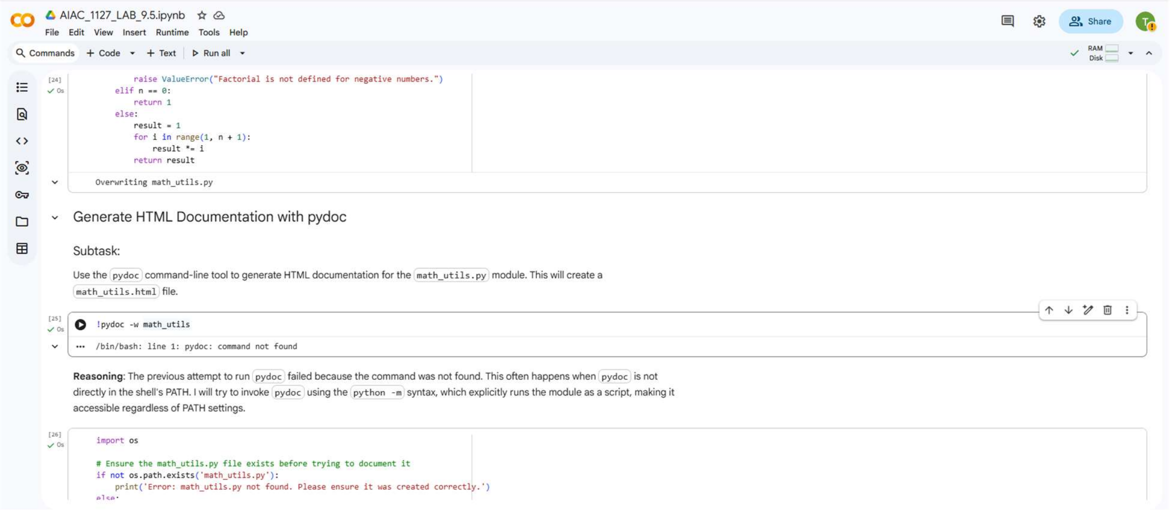
```
def check_strength(password: str) -> str:
    """Analyzes the strength of a given password.
    This function evaluates a password string and categorizes its strength as
    'Weak', 'Medium', or 'Strong' based on predefined criteria such as length
    and the presence of different character types (e.g., digits).
    Args:
        password (str): The password string to be analyzed.
    Returns:
        str: A string indicating the password's strength ('Weak', 'Medium', 'Strong').
    Examples:
        >>> check_strength("short1")
        'Weak'
        >>> check_strength("MediumP@ss")
        'Medium'
        >>> check_strength("StrongPassword123!")
        'Strong'
        >>> check_strength("12345")
        'Weak'
        >>> check_strength("onlyletters")
        'Medium'
    """
    # Placeholder logic for demonstration purposes
    if len(password) < 6:
        return "Weak"
    elif len(password) < 12 or not any(char.isdigit() for char in password):
        return "Medium"
    else:
        return "Strong"
# Example usage:
print(check_strength("test"))
print(check_strength("MyP@ssword"))
print(check_strength("VeryStrongP@ssword123"))
```

```
Weak
Medium
Strong
```

TASK 3: Math Utilities Module

Prompt : Generate a complete Google Colab workflow for creating a Python module called math_utils.py with square, cube, and factorial functions including proper docstrings. Also include the commands to save the file and generate HTML documentation using pydoc.

CODE & OUTPUT:


AIAC_1127_LAB_9.5.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM
Disk
[24]
0s
[25]
0s
[26]
0s

```
        raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
```

```
Overwriting math_utils.py
```

Generate HTML Documentation with pydoc

Subtask:

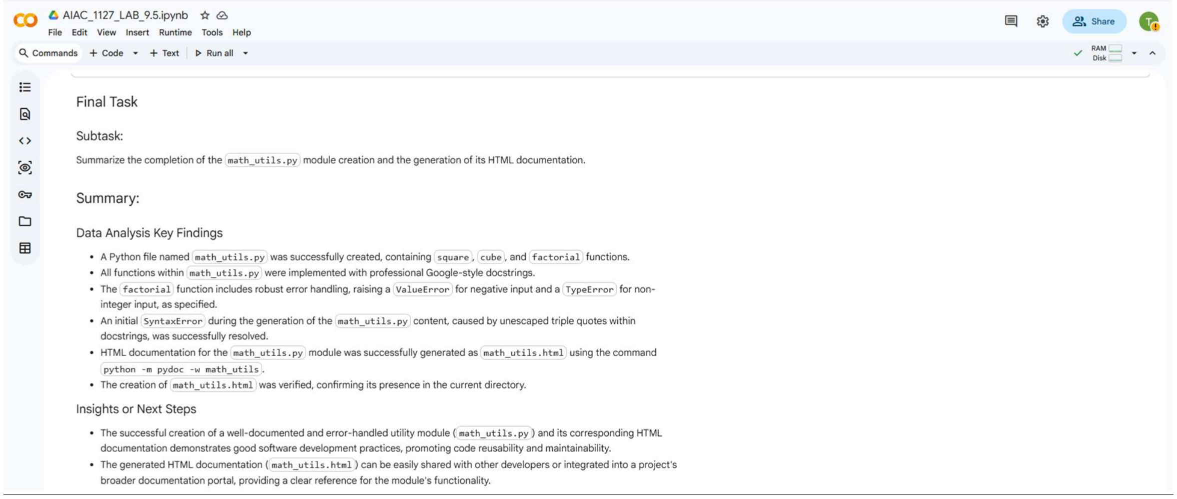
Use the `pydoc` command-line tool to generate HTML documentation for the `math_utils.py` module. This will create a `math_utils.html` file.

```
!pydoc -w math_utils
```

```
/bin/bash: line 1: pydoc: command not found
```

Reasoning: The previous attempt to run `pydoc` failed because the command was not found. This often happens when `pydoc` is not directly in the shell's PATH. I will try to invoke `pydoc` using the `python -m` syntax, which explicitly runs the module as a script, making it accessible regardless of PATH settings.

```
import os

# Ensure the math_utils.py file exists before trying to document it
if not os.path.exists('math_utils.py'):
    print('Error: math_utils.py not found. Please ensure it was created correctly.')
else:
```


AIAC_1127_LAB_9.5.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM
Disk

Final Task

Subtask:

Summarize the completion of the `math_utils.py` module creation and the generation of its HTML documentation.

Summary:

Data Analysis Key Findings

- A Python file named `math_utils.py` was successfully created, containing `square`, `cube`, and `factorial` functions.
- All functions within `math_utils.py` were implemented with professional Google-style docstrings.
- The `factorial` function includes robust error handling, raising a `ValueError` for negative input and a `TypeError` for non-integer input, as specified.
- An initial `SyntaxError` during the generation of the `math_utils.py` content, caused by unescaped triple quotes within docstrings, was successfully resolved.
- HTML documentation for the `math_utils.py` module was successfully generated as `math_utils.html` using the command `python -m pydoc -w math_utils`.
- The creation of `math_utils.html` was verified, confirming its presence in the current directory.

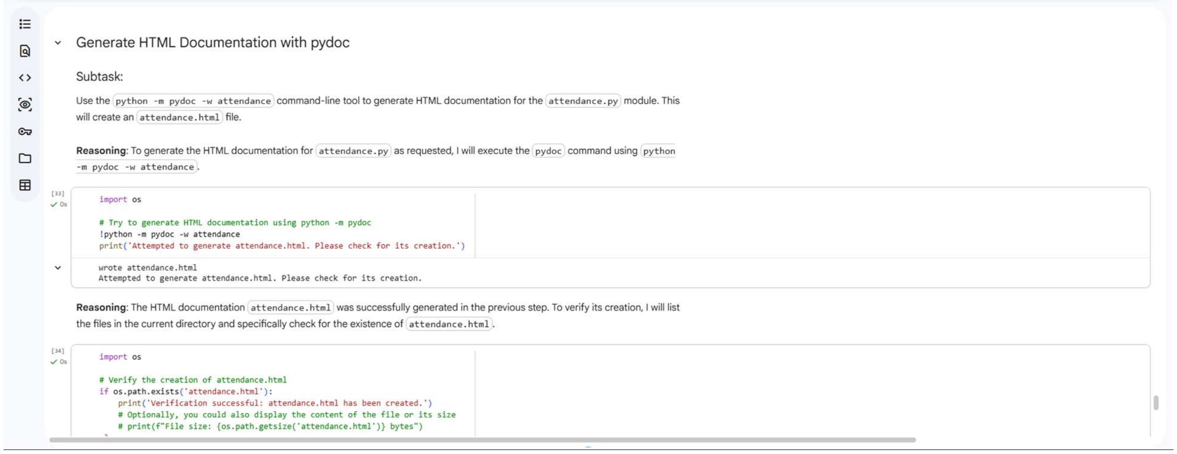
Insights or Next Steps

- The successful creation of a well-documented and error-handled utility module (`math_utils.py`) and its corresponding HTML documentation demonstrates good software development practices, promoting code reusability and maintainability.
- The generated HTML documentation (`math_utils.html`) can be easily shared with other developers or integrated into a project's broader documentation portal, providing a clear reference for the module's functionality.

TASK 4: Attendance Management Module

Prompt : Generate a Python module named attendance.py with functions mark_present(student), mark_absent(student), and get_attendance(student) using a dictionary to store records, include professional docstrings explaining purpose, parameters and return values, and also provide Google Colab commands to save the file and generate HTML documentation using pydoc.

CODE & OUTPUT:



Generate HTML Documentation with pydoc

Subtask:

Use the `python -m pydoc -w attendance` command-line tool to generate HTML documentation for the `attendance.py` module. This will create an `attendance.html` file.

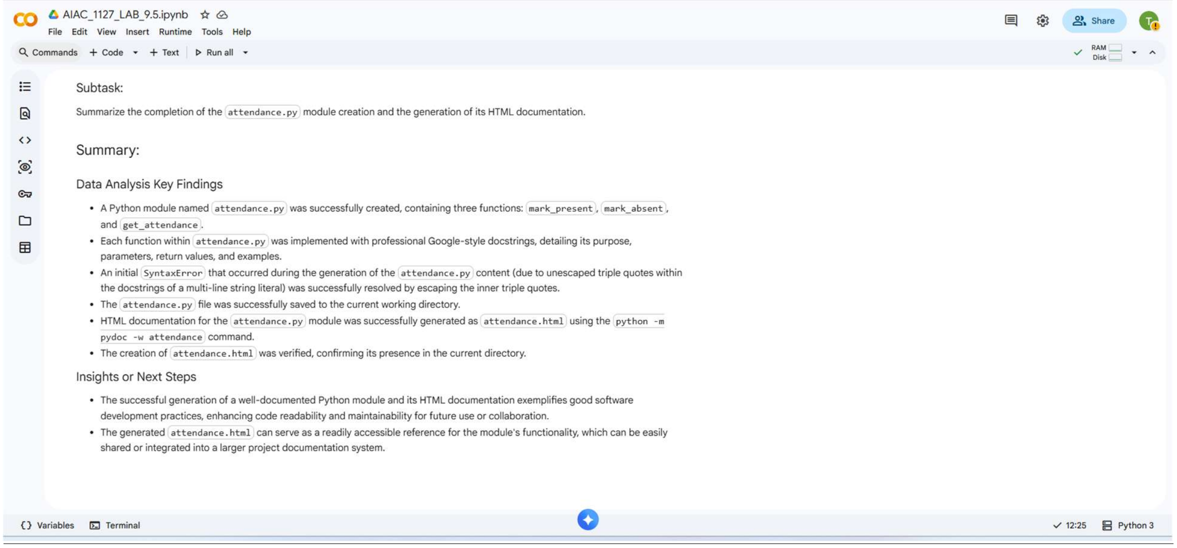
Reasoning: To generate the HTML documentation for `attendance.py` as requested, I will execute the `pydoc` command using `python -m pydoc -w attendance`.

```
import os

# Try to generate HTML documentation using python -m pydoc
!python -m pydoc -w attendance
print('Attempted to generate attendance.html. Please check for its creation.')
```

```
wrote attendance.html
Attempted to generate attendance.html. Please check for its creation.
```

Reasoning: The HTML documentation `attendance.html` was successfully generated in the previous step. To verify its creation, I will list the files in the current directory and specifically check for the existence of `attendance.html`.

```
import os

# Verify the creation of attendance.html
if os.path.exists('attendance.html'):
    print('Verification successful: attendance.html has been created.')
    # Optionally, you could also display the content of the file or its size
    # print(f"File size: {os.path.getsize('attendance.html')} bytes")
```



AIAC_1127_LAB_9.5.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM Disk
Variables Terminal
12:25 Python 3

Subtask:

Summarize the completion of the `attendance.py` module creation and the generation of its HTML documentation.

Summary:

Data Analysis Key Findings

- A Python module named `attendance.py` was successfully created, containing three functions: `mark_present`, `mark_absent`, and `get_attendance`.
- Each function within `attendance.py` was implemented with professional Google-style docstrings, detailing its purpose, parameters, return values, and examples.
- An initial `SyntaxError` that occurred during the generation of the `attendance.py` content (due to unescaped triple quotes within the docstrings of a multi-line string literal) was successfully resolved by escaping the inner triple quotes.
- The `attendance.py` file was successfully saved to the current working directory.
- HTML documentation for the `attendance.py` module was successfully generated as `attendance.html` using the `python -m pydoc -w attendance` command.
- The creation of `attendance.html` was verified, confirming its presence in the current directory.

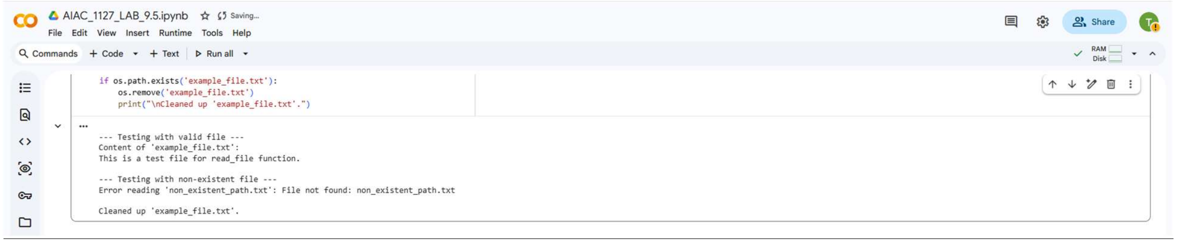
Insights or Next Steps

- The successful generation of a well-documented Python module and its HTML documentation exemplifies good software development practices, enhancing code readability and maintainability for future use or collaboration.
- The generated `attendance.html` can serve as a readily accessible reference for the module's functionality, which can be easily shared or integrated into a larger project documentation system.

TASK 5: File Handling Function

Prompt 1: Generate a PEP 257 compliant docstring for the read_file function, detailing its purpose, arguments, return value, and clearly mentioning FileNotFoundError and IOError.

OUTPUT:

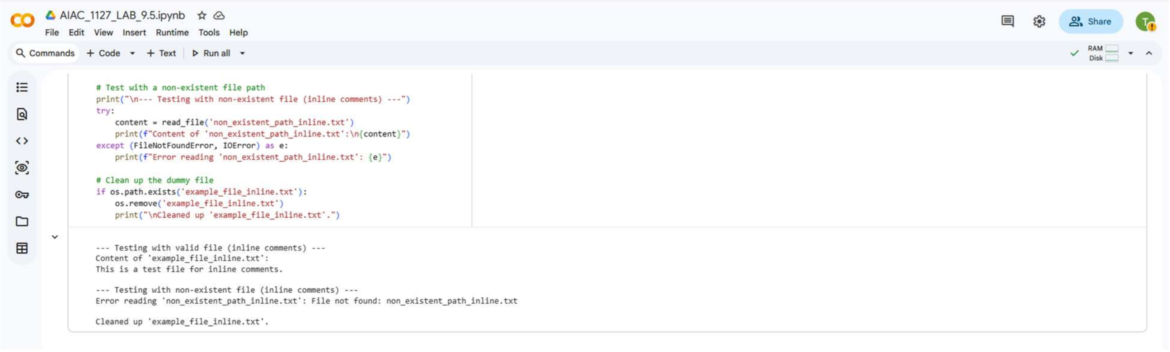


```python
if os.path.exists('example_file.txt'):
    os.remove('example_file.txt')
    print("\nCleaned up 'example_file.txt'.")
```

```
--- Testing with valid file ---
Content of 'example_file.txt':
This is a test file for read_file function.

--- Testing with non-existent file ---
Error reading 'non_existent_path.txt': File not found: non_existent_path.txt

Cleaned up 'example_file.txt'.
```

Prompt 2: Add appropriate inline comments to the read_file function to explain each line or logical block of code, including potential exception points.

CODE & OUTPUT:

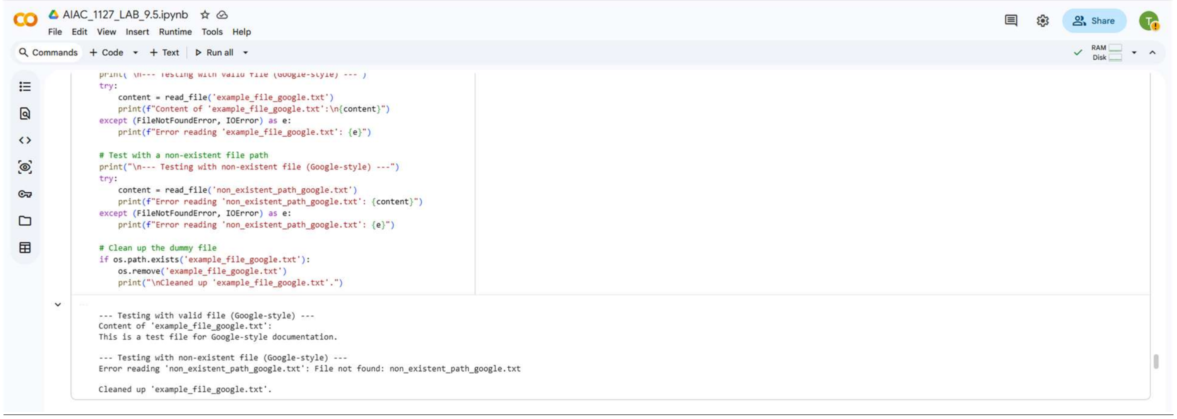


```python
# Test with a non-existent file path
print("\n--- Testing with non-existent file (inline comments) ---")
try:
    content = read_file('non_existent_path_inline.txt')
    print(f"Content of 'non_existent_path_inline.txt':\n{content}")
except (FileNotFoundError, IOError) as e:
    print(f"Error reading 'non_existent_path_inline.txt': {e}")

# Clean up the dummy file
if os.path.exists('example_file_inline.txt'):
    os.remove('example_file_inline.txt')
    print("\nCleaned up 'example_file_inline.txt'.")
```

```
--- Testing with valid file (inline comments) ---
Content of 'example_file_inline.txt':
This is a test file for inline comments.

--- Testing with non-existent file (inline comments) ---
Error reading 'non_existent_path_inline.txt': File not found: non_existent_path_inline.txt

Cleaned up 'example_file_inline.txt'.
```

Prompt 3: Generate Google-style documentation for the read_file function, including sections for arguments, returns, Raises (explicitly listing FileNotFoundError and IOError), and examples.

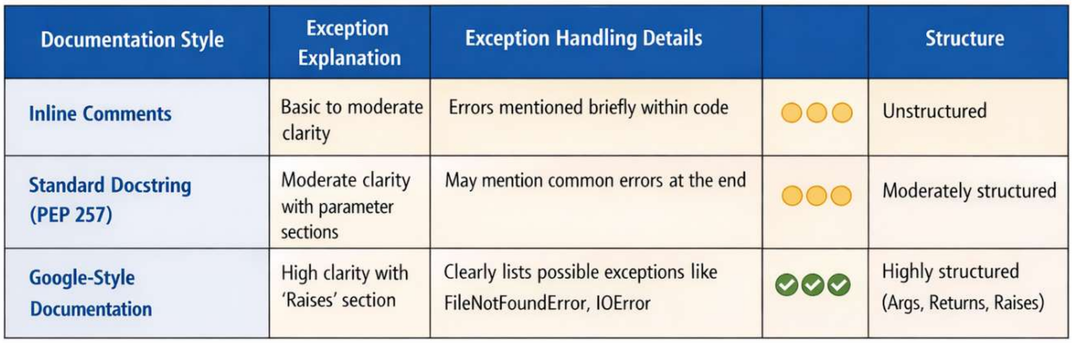
CODE & OUTPUT:



```python
print("\n--- Testing with valid file (Google-style) ---")
try:
    content = read_file('example_file_google.txt')
    print(f"Content of 'example_file_google.txt':\n{content}")
except (FileNotFoundError, IOError) as e:
    print(f"Error reading 'example_file_google.txt': {e}")

# Test with a non-existent file path
print("\n--- Testing with non-existent file (Google-style) ---")
try:
    content = read_file('non_existent_path_google.txt')
    print(f"Error reading 'non_existent_path_google.txt': {content}")
except (FileNotFoundError, IOError) as e:
    print(f"Error reading 'non_existent_path_google.txt': {e}")

# Clean up the dummy file
if os.path.exists('example_file_google.txt'):
    os.remove('example_file_google.txt')
    print("\nCleaned up 'example_file_google.txt'.")
```

```
--- Testing with valid file (Google-style) ---
Content of 'example_file_google.txt':
This is a test file for Google-style documentation.

--- Testing with non-existent file (Google-style) ---
Error reading 'non_existent_path_google.txt': File not found: non_existent_path_google.txt

Cleaned up 'example_file_google.txt'.
```

COMPARISON:



Documentation Style	Exception Explanation	Exception Handling Details		Structure
Inline Comments	Basic to moderate clarity	Errors mentioned briefly within code	●●●	Unstructured
Standard Docstring (PEP 257)	Moderate clarity with parameter sections	May mention common errors at the end	●●●	Moderately structured
Google-Style Documentation	High clarity with 'Raises' section	Clearly lists possible exceptions like FileNotFoundError, IOError	✓✓✓	Highly structured (Args, Returns, Raises)

RECOMMENDATION:

Google-style documentation is the most appropriate style for file handling functions because it clearly explains exception handling using a structured format. It provides separate sections such as Args, Returns, and Raises, which make it easy to understand possible errors like FileNotFoundError and IOError.

Since file operations are prone to runtime errors, clearly documenting exceptions improves code reliability, maintainability, and debugging. Therefore, Google-style documentation is recommended for explaining exception handling in file handling functions.