School of Computer Science and Artificial Intelligence

LabAssignment-9.5

Program : B.Tech(CSE)

Specialization : AIML

Course Title :AI Assisted Coding

Name of Student : B.Thanmai

Enollment No. : 2303A52126

Batch No. : 33

Date :13-2-2026

Problem 1: String Utilities Function

Given code:

```
def reverse_string(text):
    return text[::-1]
```

prompt:

Generate documentation for the given Python function.
Include:
1. A simple docstring
2. Inline comments
3. Google-style documentation
Also explain which documentation style is best for a string utility library and why.

Task:

1.Inline Comments:

```python
def reverse_string(text):
    # Use slicing to reverse the string efficiently.
    # [::-1] creates a reversed copy of the string.
    return text[::-1] # Return the reversed string
```

2. Docstring Documentation:

```python
def reverse_string(text):
    """Reverses the input string. Returns the reversed string."""
    return text[::-1]
```

3. Google-style Documentation:

```python
def reverse_string(text):
    """Reverses a given string.

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

    Args:
        text (str): The string to be reversed.

    Returns:
        str: The reversed string.

    Examples:
        >>> reverse_string("hello")
        "olleh"
        >>> reverse_string("world")
        "dlrow"
    """
    return text[::-1]
```

2.Compare the three documentation styles.

Comparison of Documentation Styles

Style	Description	Advantage	Limitation
Inline Comments	Comments inside code	Simple	Not reusable
Docstring	Inside function	Standard	Less structured
Google-Style	Structured format	Clear & professional	Slightly long

Recommended Style: Google-style documentation

3. Recommend the most suitable style for a utility-based string library.

API Clarity and Completeness:
A library is designed to be used by other developers. Google-style docstrings clearly define the function's contract: what parameters it expects, their types, what it returns, its type, and any potential side effects

or exceptions. This is crucial for library users to understand and correctly integrate your functions without needing to read the source code.

Automated Documentation Generation: Tools like Sphinx (with the numpydoc or sphinx.ext.napoleon extensions) can parse Google-style docstrings to automatically generate professional, navigable API documentation websites. This significantly reduces the manual effort of maintaining separate documentation and ensures it stays synchronized with the code.

Consistency: Adopting a structured style like Google-style enforces consistency across the entire library. This makes it easier for contributors to understand existing code and write new code that conforms to the same documentation standards.

Examples for Easy Adoption:
Google-style often includes example usage within the docstring. For a utility library, concrete examples of how to use each function are invaluable for quick understanding and adoption by users
.
Maintainability:
While initially more verbose, the structured nature makes it easier to review, update, and ensure completeness of the documentation as the library evolves. It prevents critical information from being missed.

Problem 2: Password Strength Checker:

Given code:

```
def check_strength(password):
return len(password) >= 8
```

prompt:

First compare documentation styles for security-related code.
Then generate:
1. Inline comments
2. Docstring
3. Google-style documentation

Finally recommend the best documentation style.

Function:

```
def check_strength(password):
    return len(password) >= 8
```

Task:

1.Inline Comments:

```
def check_strength(password):
    # The minimum length for a secure password is set to 8 characters.
    # This is a common industry standard to mitigate brute-force attacks.
    return len(password) >= 8 # Returns True if length meets requirement, False otherwise.
```

2. Docstring Documentation:

```
def check_strength(password):
    """Verifies if the provided password meets basic strength requirements."""
    return len(password) >= 8
```

3.Google-style Documentation:

```
def check_strength(password):
    """Checks if a password meets the minimum length requirement.

    This function evaluates the provided password against a predefined
    minimum length for security purposes. It is a preliminary check
    and should be combined with other strength criteria (e.g., complexity,
    entropy) for robust password policies.

    Args:
        password (str): The password string to be evaluated.

    Returns:
        bool: True if the password's length is 8 characters or more;
              False otherwise.

    Raises:
        TypeError: If `password` is not a string.

    Security Considerations:
        - This function only checks length. It does not validate for
          common passwords, character diversity, or other common
          strength metrics.
        - An empty string will return False, as its length is 0.

    Examples:
        >>> check_strength("securepwd")
        True
        >>> check_strength("short")
        False
```

```
Examples:
    >>> check_strength("securepwd")
    True
    >>> check_strength("short")
    False
"""
if not isinstance(password, str):
    raise TypeError("Password must be a string.")
return len(password) >= 8
```

2. Compare documentation styles for security-related code.

Comparison of Documentation Styles

Style	Suitability
Inline Comments	Not secure enough
Docstring	Better
Google-Style	Best

Recommended Style: Google-style documentation

3. Recommend the most appropriate style.

For Google-style documentation (or a similar comprehensive, structured style like reStructuredText or NumPy style) is the unequivocally best choice.

Precision and Unambiguity:
Security functions demand absolute clarity. Google-style docstrings force developers to explicitly define every aspect: what input password means (e.g., cleartext, hashed?), its type, the exact criteria (e.g., 'length >= 8'),

Auditability and Review:
Comprehensive documentation makes it significantly easier for security auditors, penetration testers, and peer reviewers to understand the function's security posture, identify potential flaws, and verify compliance with security requirements.

Clear API Contract:
For functions dealing with sensitive data or operations, the documentation is the contract. Google-style clearly outlines how to interact with the function securely and what security guarantees (or lack thereof) it provides. Handling Edge Cases and Errors: Security bugs often stem from mishandling unexpected inputs. Structured docstrings encourage documenting how edge cases, invalid inputs, or exceptional conditions are managed, which is crucial for robust security.

Automated Documentation Generation:
Tools can parse these structured docstrings to generate comprehensive security documentation, ensuring that the critical security information is consistently available and easily navigable.

Problem 3: Math Utilities Module:

Prompt :

Create a Python module math_utils.py.
Add functions:
1. square(n)
2. cube(n)
3. factorial(n)

Include Google-style documentation.
Explain how to generate HTML documentation.

Task:

1. Create a module math_utils.py with functions:
o square(n)
o cube(n)
o factorial(n)

```
def square(num):
    """Calculates the square of a given number.

    This function takes a number and returns its square.

    Args:
        num (int or float): The number to be squared.

    Returns:
        int or float: The square of the input number.

    Examples:
        >>> square(5)
        25
        >>> square(2.5)
        6.25
    """
    return num * num
```

```python
def cube(num):
    """Calculates the cube of a given number.

    This function takes a number and returns its cube.

    Args:
        num (int or float): The number to be cubed.

    Returns:
        int or float: The cube of the input number.

    Examples:
        >>> cube(3)
        27
        >>> cube(2.0)
        8.0
    """
    return num * num * num
```

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

    The factorial of a non-negative integer `n`, denoted by `n!`, is
    the product of all positive integers less than or equal to `n`.
    The factorial of 0 is defined as 1.

    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of `n`.

    Raises:
        ValueError: If `n` is a negative integer.

    Examples:
        >>> factorial(0)
        1
        >>> factorial(1)
        1
        >>> factorial(5)
        120
        >>> factorial(3)
        6
```

```
if not isinstance(n, int):
    raise TypeError("Input must be an integer.")
if n < 0:
    raise ValueError("Factorial is not defined for negative numbers.")
if n == 0:
    return 1
result = 1
for i in range(1, n + 1):
    result *= i
return result
```

2.Generate docstrings automatically using AI tools.

Contextual Understanding:
AI models are trained on vast amounts of code and corresponding documentation. They learn patterns in how functions, classes, and methods are typically documented.

Input Analysis:
When you feed your code to an AI docstring generator, it analyzes the function signature (name, parameters, return types), comments within the function, and sometimes even the logic inside the function body.

Docstring Suggestion:
Based on this analysis and its training, the AI suggests a docstring that summarizes the function's purpose, describes its arguments, return values, and sometimes even provides examples or raises sections.

Common Approaches and Tools:

IDE Extensions: Many popular Integrated Development Environments (IDEs) like VS Code, PyCharm, or even Jupyter environments (through specific extensions) offer AI-powered docstring generation. Examples include extensions that integrate with services like GitHub Copilot or dedicated docstring generators.

Dedicated AI Services/Libraries:
There are also standalone tools or Python libraries that focus specifically on docstring generation, often integrating with large language models.
Example of AI-Generated Docstring for square(num).

3. Export documentation as an HTML file.

Step 1: Install Sphinx and necessary extensions

First, you need to install Sphinx and the extensions required to parse Google-style docstrings and use a modern theme. You will use `pip` for this.

```
!pip install sphinx sphinx_rtd_theme sphinx_rtd_theme sphinx.ext.napoleon
```

Step 2: Set up Sphinx project

Next, you'll initialize a Sphinx project. This involves creating a documentation directory and running the `sphinx-quickstart` utility.

a. **Create a new directory for your documentation:** Create a directory named `docs` at the same level as your `math_utils.py` file. For example:

```
# Assuming you are in the parent directory where math_utils.py is located
!mkdir docs
```

b. **Navigate into this `docs` directory:**

```
%cd docs
```

c. **Run the `sphinx-quickstart` command:** This interactive script will guide you through the initial setup. When prompted, accept most defaults (e.g., separate build and source directories) but ensure you enable the `autodoc` and `napoleon` extensions. You can usually do this by typing `y` when asked. Provide a project name (e.g., `Math Utils`), your author name, and a project release (e.g., `1.0`).

```
!sphinx-quickstart
```

- When asked `Separate source and build directories (y/n)? [y]:` press `Enter` (for `y`).
- When asked `Project name:` enter `Math Utils`.
- When asked `Author name(s):` enter your name.
- When asked `Project release []:` enter `1.0`.
- When asked `Do you want to use the 'epub' builder (y/n)? [n]:` press `Enter` (for `n`).
- When asked `Do you want to use the 'autodoc' extension to enable automatic documentation of Python modules (y/n)? [n]:` type `y` and press `Enter`.
- When asked `Do you want to use the 'napoleon' extension to enable parsing of NumPy and Google style docstrings (y/n)? [n]:` type `y` and press `Enter`.
- You can accept defaults for other extensions unless you have specific needs.

Step 3: Configure `conf.py`

The `conf.py` file is the main configuration file for Sphinx. You need to adjust it to correctly locate your Python module and apply the chosen theme.

a. **Open the `conf.py` file:** It's located in your `docs/source` directory. You will need to edit this file.

b. **Uncomment and modify the `sys.path` lines:** This step is crucial for Sphinx to find your `math_utils.py` module. You need to tell Sphinx where your Python code resides relative to the `conf.py` file. Assuming `math_utils.py` is one directory up from `docs`, and `conf.py` is in `docs/source`, the path to the project root would be `../../`.

Find these lines (or similar) and uncomment them, then adjust the path:

```
import os
import sys
sys.path.insert(0, os.path.abspath('../../')) # Adjust path to point to the directory containing math_utils.py
```

c. **Ensure `autodoc`, `napoleon`, and `sphinx_rtd_theme` are in `extensions`:** Make sure the `extensions` list in `conf.py` includes these:

```
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.napoleon',
    'sphinx_rtd_theme'
]
```

d. **Set the HTML theme:** Find the `html_theme` variable and set it to `'sphinx_rtd_theme'`:

```
html_theme = 'sphinx_rtd_theme'
```

Step 4: Create ReStructuredText (RST) files for modules

Now, you'll create an RST file to instruct Sphinx on which module to document.

a. **In the `docs/source` directory, create a new RST file:** Name it `math_utils.rst`.

b. **Add content to this file:** This tells Sphinx to document the `math_utils` module and include all its members.

```
math_utils Module
=================

.. automodule:: math_utils
   :members:
```

Step 5: Include the new RST file in `index.rst`

To make your `math_utils.rst` page accessible from the main documentation, you need to link it in `index.rst`.

a. **Open `index.rst`** in the `docs/source` directory.

b. **Add `math_utils` to the `.. toctree::` directive:** This links your `math_utils.rst` file into the main documentation structure.

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   math_utils
```

Step 6: Build the HTML documentation

With all configurations in place, you can now build the HTML documentation.

a. **Navigate to the `docs` directory:** (the one created in step 2a).

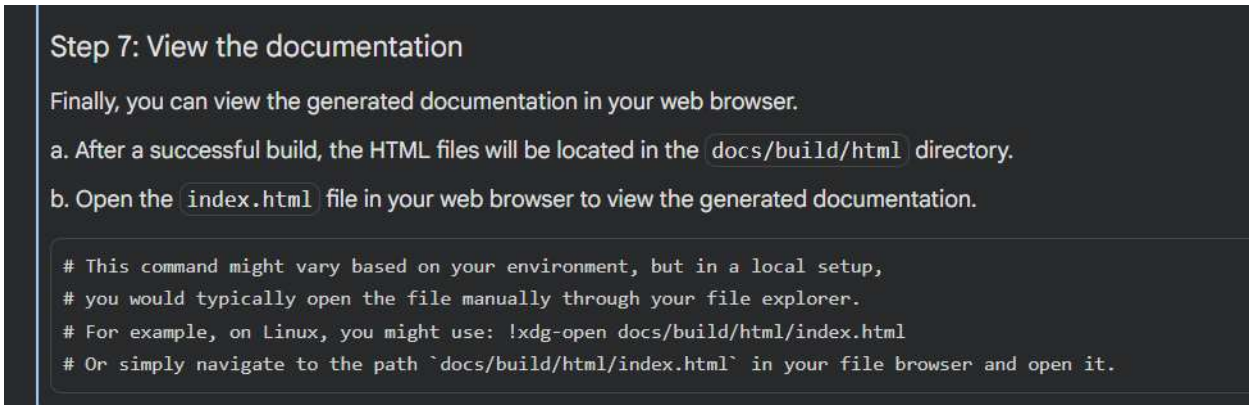
```
%cd .. # If you are still in docs/source, go back to docs
```

b. **Run the command `make html`:** This will process your RST files and generate the HTML output.

```
!make html
```

Step 7: View the documentation

Finally, you can view the generated documentation in your web browser.



Problem 4: Attendance Management Module

Prompt:

Create attendance.py with:
- mark_present(student)
- mark_absent(student)
- get_attendance(student)

Add docstrings and explain documentation viewing

Task:

1. Create a module attendance.py with functions:
-mark_present(student)
- mark_absent(student)
- get_attendance(student)

```python
# attendance.py

attendance_records = {}

def mark_present(student):
    """Marks a student as present.

    This function updates the attendance record for the specified student,
    setting their status to 'Present'.

    Args:
        student (str): The name of the student to mark as present.

    Examples:
        >>> mark_present("Alice")
        >>> get_attendance("Alice")
        'Present'
    """
    attendance_records[student] = 'Present'
    print(f"{student} marked as Present.")

def mark_absent(student):
    """Marks a student as absent.

    This function updates the attendance record for the specified student,
    setting their status to 'Absent'.

    Args:
        student (str): The name of the student to mark as absent.

    Examples:
        >>> mark_absent("Bob")
        >>> get_attendance("Bob")
        'Absent'
    """
    attendance_records[student] = 'Absent'
    print(f"{student} marked as Absent.")
```

```python
def get_attendance(student):
    """Retrieves the attendance status of a student.

    This function checks the current attendance record and returns the
    status of the specified student. If the student is not found in the
    records, it returns 'Not Found'.

    Args:
        student (str): The name of the student whose attendance status
                       is to be retrieved.

    Returns:
        str: The attendance status ('Present', 'Absent', or 'Not Found').

    Examples:
        >>> mark_present("Charlie")
        >>> get_attendance("Charlie")
        'Present'
        >>> get_attendance("David")
        'Not Found'
    """
    return attendance_records.get(student, 'Not Found')
```

2. Add proper docstrings.

Once you have defined your functions with docstrings, you can easily view this documentation directly within a Python environment (like this Colab notebook or a local Python interpreter).

Here are the primary ways to view docstrings:

1. **Using `help()` function:** The built-in `help()` function provides an interactive help system. When called with a function, module, or class name, it displays its docstring along with other useful information like its signature.

   To see the docstring for `mark_present`, you would run:

   ```
   help(mark_present)
   ```

2. **Accessing the `__doc__` attribute:** Every Python object (functions, classes, modules) that has a docstring will store it in its `__doc__` attribute.

   To see just the docstring string for `mark_absent`, you would run:

   ```
   print(mark_absent.__doc__)
   ```

3. **Using `?` in IPython/Colab:** In IPython environments (which Colab uses), you can use a question mark `?` after a function name to quickly display its docstring and other information.

   For `get_attendance`, you would type:

   ```
   get_attendance?
   ```

```
help(mark_present)
```

```
Help on function mark_present in module __main__:

mark_present(student)
    Marks a student as present.

    This function updates the attendance record for the specified student,
    setting their status to 'Present'.

    Args:
        student (str): The name of the student to mark as present.

    Examples:
        >>> mark_present("Alice")
        >>> get_attendance("Alice")
        'Present'
```

```
print(get_attendance.__doc__)
```

```
Retrieves the attendance status of a student.

    This function checks the current attendance record and returns the
    status of the specified student. If the student is not found in the
    records, it returns 'Not Found'.

    Args:
        student (str): The name of the student whose attendance status
                       is to be retrieved.

    Returns:
        str: The attendance status ('Present', 'Absent', or 'Not Found').

    Examples:
        >>> mark_present("Charlie")
        >>> get_attendance("Charlie")
        'Present'
        >>> get_attendance("David")
        'Not Found'
```

3. Generate and view documentation in terminal and browse

Install Sphinx and Dependencies

Subtask:

Install Sphinx, the `sphinx_rtd_theme`, and `sphinx.ext.napoleon` using pip.

Reasoning: The user wants to install Sphinx and its necessary extensions. A code cell with a `pip install` command is required for this.

```
pip install sphinx sphinx_rtd_theme sphinx.ext.napoleon
```

```
Requirement already satisfied: sphinx in /usr/local/lib/python3.12/dist-packages (8.2.3)
Collecting sphinx_rtd_theme
  Downloading sphinx_rtd_theme-3.1.0-py2.py3-none-any.whl.metadata (4.5 kB)
ERROR: Could not find a version that satisfies the requirement sphinx.ext.napoleon (from versions: none)
ERROR: No matching distribution found for sphinx.ext.napoleon
```

Reasoning: The previous installation command failed because `sphinx.ext.napoleon` is not a package that can be installed directly via pip. It is an extension within Sphinx that is enabled in the configuration. I will re-run the `pip install` command to correctly install `sphinx` and `sphinx_rtd_theme`.

```
pip install sphinx sphinx_rtd_theme
```

```
Requirement already satisfied: sphinx in /usr/local/lib/python3.12/dist-packages (8.2.3)
Collecting sphinx_rtd_theme
  Using cached sphinx_rtd_theme-3.1.0-py2.py3-none-any.whl.metadata (4.5 kB)
Requirement already satisfied: sphinxcontrib-applehelp>=1.0.7 in /usr/local/lib/python3.12/dist-packages (from sphinx)
Requirement already satisfied: sphinxcontrib-devhelp>=1.0.6 in /usr/local/lib/python3.12/dist-packages (from sphinx) (
```

Subtask:

Set up the Sphinx project by creating a `docs` directory and running `sphinx-quickstart` inside it. Ensure you enable `autodoc` and `napoleon` when prompted.

Reasoning: First, I will create a `docs` directory and navigate into it, as required for setting up the Sphinx project. This is a prerequisite for running `sphinx-quickstart`.

```
import os

# Get the current working directory
current_dir = os.getcwd()

# Define the path for the new 'docs' directory
docs_dir = os.path.join(current_dir, 'docs')

# Create the 'docs' directory if it doesn't exist
if not os.path.exists(docs_dir):
    os.makedirs(docs_dir)
    print(f"Created directory: {docs_dir}")
else:
    print(f"Directory already exists: {docs_dir}")

# Change the current working directory to 'docs'
os.chdir(docs_dir)
print(f"Changed current working directory to: {os.getcwd()}")
```

```
Created directory: /content/docs
Changed current working directory to: /content/docs
```

```python
from subprocess import Popen, PIPE

# Define the expected inputs for sphinx-quickstart
# y for separate source/build, Project name, Author name, Project release,
# y for autodoc, y for napoleon, and then Enter for the rest of the defaults.
inputs = b'y\nMath Utils\nYour Name\n1.0\ny\ny\n\n\n\n\n\n'

# Run sphinx-quickstart with the predefined inputs
# We use Popen and communicate to feed inputs to the interactive process
print("Running sphinx-quickstart interactively...")
process = Popen(['sphinx-quickstart'], stdin=PIPE, stdout=PIPE, stderr=PIPE)
stdout, stderr = process.communicate(input=inputs)

print("sphinx-quickstart output:")
print(stdout.decode())
if stderr:
    print("sphinx-quickstart errors:")
    print(stderr.decode())

# List the created files to confirm setup
print("\nFiles created in docs directory:")
!ls -F
```

```
Running sphinx-quickstart interactively...
sphinx-quickstart output:
Welcome to the Sphinx 8.2.3 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: .

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
```

Configure `conf.py`

Subtask:

Modify the `conf.py` file to set the Python path, include `autodoc`, `napoleon`, and `sphinx_rtd_theme` in extensions, and define the `html_theme`.

Instructions

1. **Open the `conf.py` file:** It's located in your `docs/source` directory. You will need to edit this file.
2. **Uncomment and modify the `sys.path` lines:** This step is crucial for Sphinx to find your `math_utils.py` module. You need to tell Sphinx where your Python code resides relative to the `conf.py` file. Assuming `math_utils.py` is one directory up from `docs` (i.e., `/content/math_utils.py`) and `conf.py` is in `docs/source` (i.e., `/content/docs/source`), the path to the project root would be `../../../` to get to `/content` where `math_utils.py` will be.

   Find these lines (or similar) and uncomment them, then adjust the path:

   ```python
   import os
   import sys
   sys.path.insert(0, os.path.abspath('../../../')) # Adjust path to point to the directory containing math_utils.py
   ```

3. **Ensure `autodoc`, `napoleon`, and `sphinx_rtd_theme` are in `extensions`:** Make sure the `extensions` list in `conf.py` includes these:

```
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.napoleon',
    'sphinx_rtd_theme'
]
```

4. **Set the HTML theme:** Find the `html_theme` variable and set it to `'sphinx_rtd_theme'`:

```
html_theme = 'sphinx_rtd_theme'
```

Create ReStructuredText (RST) files for modules

Subtask:

Create an RST file to instruct Sphinx on which module to document.

Instructions

1. **In the `docs/source` directory, create a new RST file:** Name it `math_utils.rst`.
2. **Add content to this file:** This tells Sphinx to document the `math_utils` module and include all its members.

```
math_utils Module
=================

.. automodule:: math_utils
   :members:
```

Reasoning: I need to create the `math_utils.rst` file in the `docs/source` directory with the specified content, which tells Sphinx to automatically document the `math_utils` module.

69382ff4).

- **Build HTML Documentation:** Navigate back to the `docs` directory (if you're not there) and run the `make html` command to build the HTML documentation files. The output will be visible in the terminal. Refer to Step 6 in the 'Explain HTML Documentation Generation' section (Cell 69382ff4).
- **View Generated Documentation:** Locate the generated `index.html` file within the `docs/build/html` directory and open it in your web browser to view the documentation. Refer to Step 7 in the 'Explain HTML Documentation Generation' section (Cell 69382ff4).
- **Final Task:** Confirm that the HTML documentation has been successfully generated and viewed.

Problem 5: File Handling Function:

Given Code:

```
def read_file(filename):
    with open(filename, 'r') as f:
        return f.read()
```

Prompt:

First compare documentation styles for file handling and exception handling.
Then generate:
1. Inline comments
2. Docstring
3. Google-style documentation

Recommend the best style.

Task:

Write documentation using all three formats.

1. **Inline Comments**
   - **Purpose:** To explain *how* specific lines of code work, the rationale behind certain decisions, or the mechanics of complex logic. In file handling, this could explain the purpose of `try-except` blocks or the significance of `with open()`.
   - **Example for `read_and_process_file`:**

```python
def read_and_process_file(filepath):
    # This block attempts to open and read the file securely.
    try:
        # Use 'with' statement for automatic file closing, even on error.
        with open(filepath, 'r') as file:
            content = file.read() # Read entire file content into a string.
        print(f"Successfully read file '{filepath}':\n---\n{content}\n---")
        return content # Return the file content on success.
    # Catch FileNotFoundError if the path specified doesn't point to an existing file.
    except FileNotFoundError:
        print(f"Error: The file '{filepath}' was not found.")
        return None # Indicate failure by returning None.
    # Catch broader IOError for other input/output related problems (e.g., permissions).
    except IOError as e:
        print(f"Error: An I/O error occurred while reading '{filepath}'. Details: {e}")
        return None # Indicate failure by returning None.
```

2. **Docstring Documentation (Simple/Basic)**

- **Purpose:** To provide a concise summary of what the function does. For file handling functions, this would state its primary goal of reading a file and handling I/O errors.
- **Example for `read_and_process_file`:**

```
def read_and_process_file(filepath):
    """Attempts to open and read a file, handling common I/O errors.
    Returns the content or None on failure.
    """
    try:
        with open(filepath, 'r') as file:
            content = file.read()
        print(f"Successfully read file '{filepath}':\n---\n{content}\n---")
        return content
    except FileNotFoundError:
        print(f"Error: The file '{filepath}' was not found.")
        return None
    except IOError as e:
        print(f"Error: An I/O error occurred while reading '{filepath}'. Details: {e}")
        return None
```

3. **Google-style Documentation**

- **Purpose:** A comprehensive and structured approach to documenting functions, including a summary, detailed description, arguments (with types), return values (with types), and crucially, a clear enumeration of exceptions raised. For file handling, this is ideal for explicitly stating I/O related exceptions.
- **Example for `read_and_process_file`:**

```
def read_and_process_file(filepath):
    """Attempts to open and read a file, handling common I/O errors.

    This function takes a file path, attempts to open and read the entire
    content of the file. It gracefully handles cases where the file does
    not exist or other common input/output errors occur.

    Args:
        filepath (str): The path to the file to be read.

    Returns:
        str or None: The content of the file as a string if the operation
                     is successful. Returns `None` if `FileNotFoundError`
                     or `IOError` occurs, indicating failure.

    Raises:
        # Note: The function handles these internally and returns None, but
        # for a function that *propagates* exceptions, they would be listed here.
        # If we were to re-raise them, they would be listed as:
        # FileNotFoundError: If the file specified by `filepath` does not exist.
        # IOError: For other input/output related issues (e.g., permission denied).
```

```
        >>> # Clean up dummy file
        >>> import os
        >>> os.remove('example.txt')
    """
    try:
        with open(filepath, 'r') as file:
            content = file.read()
        print(f"Successfully read file '{filepath}':\n---\n{content}\n---")
        return content
    except FileNotFoundError:
        print(f"Error: The file '{filepath}' was not found.")
        return None
    except IOError as e:
        print(f"Error: An I/O error occurred while reading '{filepath}'. Details: {e}")
        return None
```

2. Identify which style best explains exception handling.

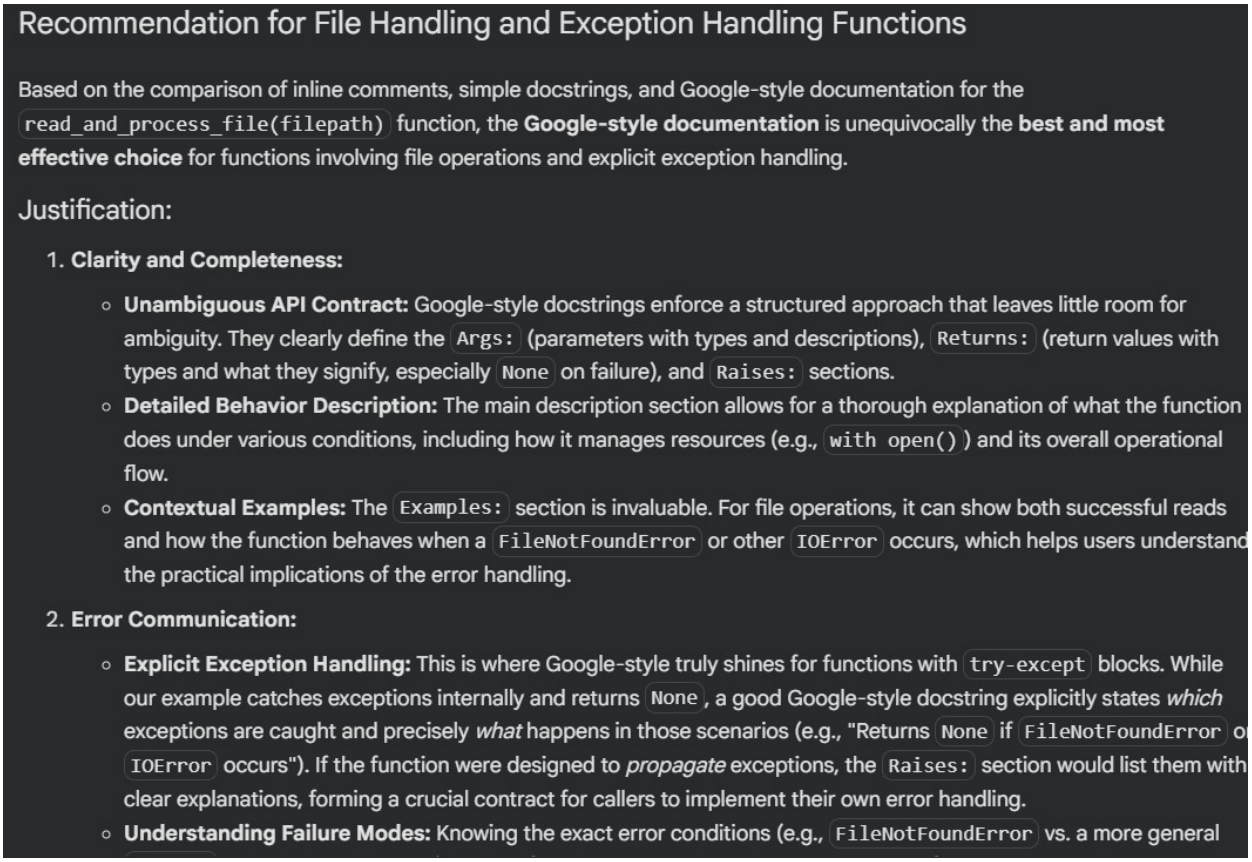
```
def read_file(filename):
    """
    Reads and returns the content of a file.

    Args:
        filename (str): Path of the file to read.

    Returns:
        str: Content of the file.

    Raises:
        FileNotFoundError: If the file does not exist.
        IOError: If the file cannot be read.
    """
    with open(filename, 'r') as f:
        return f.read()
```

3. Justify your recommendation.



Recommendation for File Handling and Exception Handling Functions

Based on the comparison of inline comments, simple docstrings, and Google-style documentation for the `read_and_process_file(filepath)` function, the **Google-style documentation** is unequivocally the **best and most effective choice** for functions involving file operations and explicit exception handling.

Justification:

1. **Clarity and Completeness:**
    - **Unambiguous API Contract:** Google-style docstrings enforce a structured approach that leaves little room for ambiguity. They clearly define the `Args:` (parameters with types and descriptions), `Returns:` (return values with types and what they signify, especially `None` on failure), and `Raises:` sections.
    - **Detailed Behavior Description:** The main description section allows for a thorough explanation of what the function does under various conditions, including how it manages resources (e.g., `with open()`) and its overall operational flow.
    - **Contextual Examples:** The `Examples:` section is invaluable. For file operations, it can show both successful reads and how the function behaves when a `FileNotFoundError` or other `IOError` occurs, which helps users understand the practical implications of the error handling.
2. **Error Communication:**
    - **Explicit Exception Handling:** This is where Google-style truly shines for functions with `try-except` blocks. While our example catches exceptions internally and returns `None`, a good Google-style docstring explicitly states *which* exceptions are caught and precisely *what* happens in those scenarios (e.g., "Returns `None` if `FileNotFoundError` o `IOError` occurs"). If the function were designed to *propagate* exceptions, the `Raises:` section would list them with clear explanations, forming a crucial contract for callers to implement their own error handling.
    - **Understanding Failure Modes:** Knowing the exact error conditions (e.g., `FileNotFoundError` vs. a more general