2303A51203
Batch 04
Assignment 9.5

Problem 1: String Utilities Function

```
def reverse_string_docstring(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 input string to be reversed.

    Returns:
        str: The reversed string.

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

# Test the function
print(f"'hello' reversed: {reverse_string_docstring('hello')}")
```

OUTPUT:

```
'hello' reversed: olleh
```

```
def reverse_string_inline(text):
    # This function reverses a given string.
    # It takes a single argument 'text', which should be a string.
    # The function uses string slicing to create a reversed copy of the
input string.
    # The slice [::-1] creates a reversed copy.
    return text[::-1] # Returns the reversed string.
```

```python
# Test the function
print(f"'world' reversed: {reverse_string_inline('world')}")
```

OUTPUT

```
'world' reversed: dlrow
```

```python
def reverse_string_google_style(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 input string to be reversed.

    Returns:
        str:
            The reversed string.

    Examples:
        >>> reverse_string_google_style("example")
        'elpmaxe'
        >>> reverse_string_google_style("12345")
        '54321'
    """
    return text[::-1]

# Test the function
print(f"'Google' reversed: {reverse_string_google_style('Google')}")
```

OUTPUT

```
'Google' reversed: elgooG
```

JUSTIFICATION:

1. **Clarity and Readability:** Google-style docstrings provide a clear, structured way to describe each function's purpose, parameters, return values, and potential exceptions. This is crucial for library users who need to quickly understand how to use each utility function without diving into the source code.
2. **Standardization:** While not a PEP standard like reStructuredText, Google-style has become a widely accepted and often preferred standard in the Python community, especially for libraries. This consistency makes it easier for developers to contribute and understand the codebase.

3. **Tooling Support:** Documentation generators like Sphinx, when combined with extensions like `sphinx.ext.autodoc` and `sphinx.ext.napoleon` (which supports Google-style), can automatically generate comprehensive API documentation from these docstrings.
4. **Examples:** The explicit `Examples:` section in Google-style is extremely beneficial for utility functions, allowing users to see practical usage immediately.

Problem 2: Password Strength Checker

CODE:

```
def check_strength_docstring(password):
    """Checks the strength of a given password.

    This function determines if a password meets a basic strength
requirement
    by checking if its length is at least 8 characters.

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

    Returns:
        bool: True if the password length is 8 or more, False otherwise.

    Examples:
        >>> check_strength_docstring("short")
        False
        >>> check_strength_docstring("secure_password")
        True
    """
    return len(password) >= 8

# Test the function
print(f"'short' strength: {check_strength_docstring('short')}")
print(f"'secure_password' strength:
{check_strength_docstring('secure_password')}")
```

OUTPUT

```
'short' strength: False
'secure_password' strength: True
```

CODE:

```
def check_strength_inline(password):
    8.
    return len(password) >= 8
```

```
print(f"'1234567' strength: {check_strength_inline('1234567')}")
```

OUTPUT:

```
'1234567' strength: False
'123456789' strength: True
def check_strength_google_style(password):
    """Checks the strength of a given password.
    This function evaluates a password based on a simple length criterion:
    it must be at least 8 characters long to be considered strong.
    Note that this is a very basic check and real-world password policies
    are typically more complex.
    Args:
        password (str):
            The password string to be evaluated.
    Returns:
        bool:
            True if the password's length is 8 or more characters; False
otherwise.
    Examples:
        >>> check_strength_google_style("abc")
        False
        >>> check_strength_google_style("MySecretPw")
        True
    """    return len(password) >= 8
# Test the function
print(f"'simple' strength: {check_strength_google_style('simple')}")
print(f"'A_Strong_Password_123' strength:
{check_strength_google_style('A_Strong_Password_123')}")
```

OUTPUT:

```
simple' strength: False
'A_Strong_Password_123' strength: True
```

Problem 3: Math Utilities Module

CODE:

```
import os
import pydoc

# Generate HTML documentation for math_utils.py
output_filename = 'math_utils.html'
pydoc.writedoc('math_utils', './')

# Rename the generated file to match the desired output_filename
```

```
# pydoc.writedoc creates a file named 'math_utils.html' by default in the
specified directory.
# We can just print a confirmation.

print(f"HTML documentation generated as '{output_filename}' in the current
directory.")
```

OUTPUT:

```
math_utils.py created with square, cube, and factorial functions.
wrote math_utils.html
HTML documentation generated as 'math_utils.html' in the current
directory.
```

Problem 4: Attendance Management Module

```
%%writefile attendance.py

attendance = {}

def mark_present(student):
    """Marks a student as 'Present' in the attendance record.

    This function updates the global `attendance` dictionary to record
    that the specified student is present.

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

    Returns:
        None

    Examples:
        >>> mark_present("Alice")
        >>> get_attendance("Alice")
        'Present'
    """
    attendance[student] = 'Present'

def mark_absent(student):
    """Marks a student as 'Absent' in the attendance record.

    This function updates the global `attendance` dictionary to record
```

```
    that the specified student is absent.

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

    Returns:
        None

    Examples:
        >>> mark_absent("Bob")
        >>> get_attendance("Bob")
        'Absent'
    """
    attendance[student] = 'Absent'

def get_attendance(student):

    return attendance.get(student, 'Not Recorded')

print("attendance.py created with mark_present, mark_absent, and
get_attendance functions.")
```

OUTPUT:

```
Writing attendance.py
```

CODE:

```
mark_absent(student)
        Marks a student as 'Absent' in the attendance record.

        This function updates the global `attendance` dictionary to record
        that the specified student is absent.
```

Problem 5: File Handling Function

```
def read_file_docstring(filename):
    """Reads the content of a specified file.

    This function attempts to open and read the entire content of a text
file.
    If the file does not exist, it prints an error message and returns
None.

    Args:
```

```
        filename (str): The path to the file to be read.

    Returns:
        str or None: The content of the file as a string, or None if an
error occurs.

    Raises:
        FileNotFoundError: If the specified file does not exist.
        IOError: For other input/output errors during file operations.

    Examples:
        >>> with open('test.txt', 'w') as f: f.write('Hello Docstring')
        >>> read_file_docstring('test.txt')
        'Hello Docstring'
        >>> read_file_docstring('non_existent_file.txt')
        'Error: The file non_existent_file.txt was not found.'
        None
    """
    try:
        with open(filename, 'r') as f:
            return f.read()
    except FileNotFoundError:
        print(f"Error: The file {filename} was not found.")
        return None
    except IOError as e:
        print(f"Error reading file {filename}: {e}")
        return None

# Test the function
with open('docstring_example.txt', 'w') as f:
    f.write('This is a test file for docstring example.')
print(f"Content: {read_file_docstring('docstring_example.txt')}")
print(f"Content: {read_file_docstring('non_existent_docstring.txt')}")
```

OUTPUT:

```
Content: This is a test file for docstring example.
Error: The file non_existent_docstring.txt was not found.
Content: None
```

CODE:

```
def read_file_inline(filename):
    # This function reads the content of a file.
```

```
    # It takes the 'filename' (string) as input.
    try: # Start a try block to handle potential file errors.
        with open(filename, 'r') as f: # Open the file in read mode ('r').
            return f.read() # Read and return the entire content of the
file.
    except FileNotFoundError: # Catch the specific error if the file
doesn't exist.
        print(f"Error: The file {filename} was not found.") # Inform the
user about the missing file.
        return None # Return None to indicate failure.
    except IOError as e: # Catch other general I/O errors.
        print(f"Error reading file {filename}: {e}") # Print the specific
I/O error.
        return None # Return None to indicate failure.

# Test the function
with open('inline_example.txt', 'w') as f:
    f.write('This is a test file for inline comments example.')
print(f"Content: {read_file_inline('inline_example.txt')}")
print(f"Content: {read_file_inline('non_existent_inline.txt')}")
```

OUTPUT:

```
Content: This is a test file for inline comments example.
Error: The file non_existent_inline.txt was not found.
Content: None
```