

AI ASSISTED CODING

Shashank

2303A510i3

BATCH – 03

13 – 02 – 2026

ASSIGNMENT – 9.5

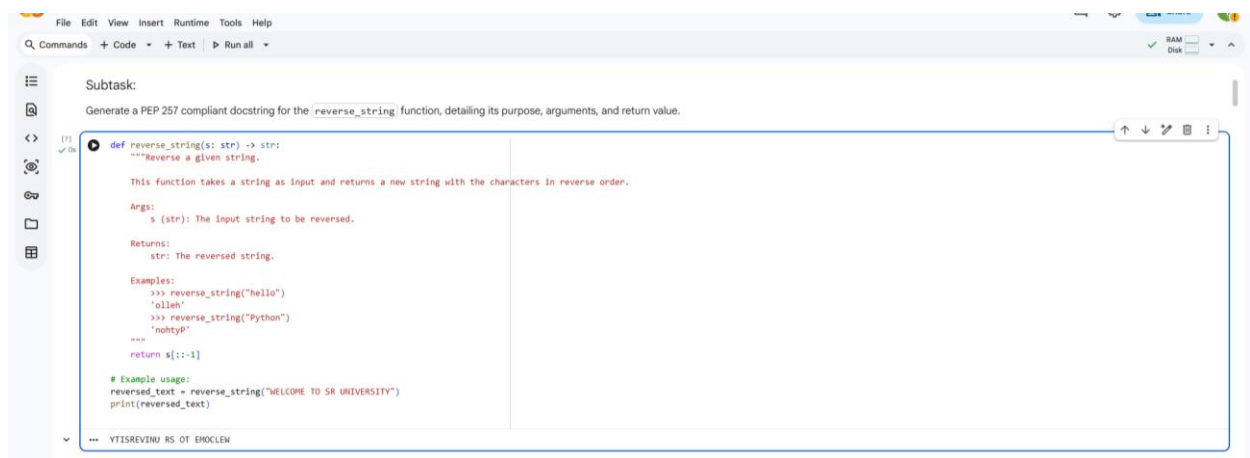
Lab 9.5: Documentation Generation -Automatic

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:



The screenshot shows a code editor with a file named 'Subtask.py'. The code defines a function `reverse_string(s: str) -> str:` with a PEP 257 compliant docstring. The docstring includes a description, arguments, returns, and examples. The function returns `s[::-1]`. An example usage is shown at the bottom.

```
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)
```

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)
```

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)
```

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:

```

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("VeryStrong@ssword123"))
  
```

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

CODE & OUTPUT:

```
File Edit View Insert Runtime Tools Help
Commands + Code + Text Runall
RAM
Out

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("Medium@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("HyP@ssword"))
print(check_strength("VeryStrong@ssword123"))

Weak
Medium
Strong
```

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

CODE & OUTPUT:

```
[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 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("Medium@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("HyP@ssword"))
print(check_strength("VeryStrong@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:

```
Q Commands + Code + Text ▶ Run all ✓ RAM Disk ^
[24] ✓ 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.
[25] ✓ 0s
!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.
[26] ✓ 0s
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:
```

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.

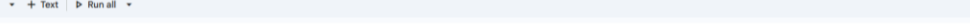
Insights or Next Steps

- 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:

[illegible]

The screenshot shows a VS Code editor window with a Python script in the main editor and its output in the terminal. The script defines a `read_file` function that checks if a file exists and prints its content or an error message. The terminal output shows the function being tested with both a valid file and a non-existent file.

```

Q Commands + Code + Text | Run all
RAM 128K

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:

```
# 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': {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("Uncleaned 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:

```
#!/usr/bin/env python3
"""
Testing with valid file (Google-style)
"""
try:
    content = read_file('example_file_google.txt')
    print(f"Content of 'example_file_google.txt': {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("Uncleaned 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 <code>FileNotFoundError</code> , <code>IOError</code> | ✓✓✓ | 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.