SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name		Venkataramana Veeramsetty	
Instructor(s) Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)	
Course Code	24CS002PC215	**Course Title**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
Date and Day of Assignment	Week5 - Monday	**Time(s)**	
Duration	2 Hours	**Applicable to Batches**	
AssignmentNumber: 9.1(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*Expected Time to complete*
1	**Lab 9 – Documentation Generation: Automatic Documentation and Code Comments** **Lab Objectives** • To use AI-assisted coding tools for generating Python documentation and code comments. • To apply zero-shot, few-shot, and context-based prompt engineering for documentation creation. • To practice generating and refining docstrings, inline comments,	Week5 - Monday

	and module-level documentation. • To compare outputs from different prompting styles for quality analysis.	

	Task Description #1 (Documentation – Google-Style Docstrings for Python Functions)
	• Task: Use AI to add Google-style docstrings to all functions in a given Python script.
	• Instructions:
	  o Prompt AI to generate docstrings without providing any input-output examples.
	  o Ensure each docstring includes:
	  ▪ Function description
	  ▪ Parameters with type hints
	  ▪ Return values with type hints
	  ▪ Example usage
	  o Review the generated docstrings for accuracy and formatting.
	• Expected Output #1:
	  o A Python script with all functions documented using correctly formatted Google-style docstrings.
	PROMPT:
	
	Each docstring should include:
	• A brief description of the function
	• Parameters with type hints (or "type" if no type hint provided)
	• Return value with type hint (or "type" if none provided)
	• An example usage section (without actual input/output)
	
	CODE:	

```
9.1.1.py > ...
def generate_docstring(func_name, params, returns):
    param_lines = ""
    for p in params:
        param_lines += f"    {p} (type): Description.\n"
    docstring = f'''"""
{func_name} function.
Args:
{param_lines}
Returns:
    {returns}: Description.

Example: ...
"""'''return docstring
def add_docstrings_to_code(code):
    lines = code.split('\n')
    new_lines = []
    inside_function = False
    func_name = ""
    params = []
    returns = "type"
    for line in lines:
        stripped = line.strip()
        if stripped.startswith("def "):
            inside_function = True
            header = stripped[4:].split('(')
            func_name = header[0].strip()
            params_str = header[1].split(')')[0]
            params = [p.split(':')[0].strip() for p in params_str.split(',') if p]
            returns = "type"
            new_lines.append(line)
            docstring = generate_docstring(func_name, params, returns)
            new_lines.append("    " + docstring.replace('\n', '\n    '))
        else:
            new_lines.append(line)
    return "\n".join(new_lines)
code_sample = '''
def add(a: int, b: int):
```

OUTPUT:

OBSERVATION:

The docstrings followed the correct formatting conventions, improving both readability and consistency across the script. This level of structured documentation helps developers quickly understand how to use a function without needing to read through its full implementation. Overall, the inclusion of properly formatted Google-style docstrings significantly enhanced the clarity, usability, and professional appearance of the code, making it easier to maintain and integrate into larger projects.

Task Description #2 (Documentation – Inline Comments for Complex Logic)

- Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.
- Instructions:

- Provide a Python script without comments to the AI.
- Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.
- Verify that comments improve code readability and maintainability.

- Expected Output #2:
  - Python code with concise, context-aware inline comments for complex logic blocks.

PROMPT:

Please add meaningful inline comments to the following Python script.
Focus only on explaining complex, tricky, or non-intuitive logic **parts,** such as nested loops, complicated conditionals, or any subtle algorithmic steps.
Do not add comments for simple or obvious syntax like variable assignments or straightforward statements.
The goal is to improve the code's readability and maintainability by clarifying only the challenging sections.

CODE:

```
def add_comments_to_complex_logic(code):
    lines = code.split('\n')
    commented_lines = []
    indent_levels = []

    for line in lines:
        stripped = line.lstrip()
        indent = len(line) - len(stripped)

        # Remove indent levels that are no longer current
        while indent_levels and indent_levels[-1] >= indent:
            indent_levels.pop()

        # If current line is a control statement and inside another block -> complex logic
        if any(stripped.startswith(kw) for kw in ['for ', 'while ', 'if ', 'elif ', 'else:']):
            if indent_levels:
                commented_lines.append(' ' * indent + '# Complex nested logic')
            indent_levels.append(indent)

        commented_lines.append(line)

    return '\n'.join(commented_lines)
code_sample = '''
def example(lst):
    total = 0
    for i in lst:
        if i % 2 == 0:
            total += i
    return total
'''

print(add_comments_to_complex_logic(code_sample))
    return '\n'.join(commented_lines)
```

OUTPUT:

OBSERVATION:
The added comments were concise, context-aware, and non-redundant, helping readers understand the *"why"* behind certain decisions without cluttering the code with obvious remarks. As a result, the script became much easier to follow and maintain.
This approach ensured that the documentation effort was both efficient and valuable, focusing developer attention only where it's needed most.

Task Description #3 (Documentation – Module-Level Documentation)

- Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.
- Instructions:
  - Supply the entire Python file to AI.
  - Instruct AI to write a single multi-line docstring at the top of the file.
  - Ensure the docstring clearly describes functionality and usage without rewriting the entire code.
- Expected Output #3:
  - A complete, clear, and concise module-level docstring at the beginning of the file.

PROMPT:

Please add a module-level docstring at the very top of the following Python file.
The docstring should be a concise multi-line summary that includes:

- The overall purpose of the module
- Any key dependencies or imported libraries
- The main functions and classes provided by the module.

CODE:

AI > 9.1.3.PY > add_module_docstring

```python
def add_module_docstring(source_code):
    docstring = '''"""
Module Summary:
This module contains functions and classes for your project.

Dependencies:
- List any dependencies here

Main Functions and Classes:
- List main functions and classes here

Usage:
Import this module and use the provided functions and classes.
"""
'''
    return docstring + '\n' + source_code

# Example usage
code = '''
def greet(name):
    print(f"Hello, {name}!")

class Person:
    def __init__(self, name):
        self.name = name
'''

updated_code = add_module_docstring(code)
print(updated_code)

```

OUTPUT:

OBSERVATION:

This high-level overview helped provide immediate orientation for developers reading the file and improved both the professionalism and maintainability of the codebase. It also made the module easier to integrate with tools that generate documentation automatically. By not duplicating code content and focusing only on the essential summary, the docstring remained concise and useful.

Task Description #4 (Documentation – Convert Comments to Structured Docstrings)

- Task: Use AI to transform existing inline comments into structured function docstrings following Google style.
- Instructions:
    - Provide AI with Python code containing inline comments.

- Ask AI to move relevant details from comments into function docstrings.
- Verify that the new docstrings keep the meaning intact while improving structure.

- Expected Output #4:
  - Python code with comments replaced by clear, standardized docstrings.

PROMPT:

Please convert all inline comments inside functions in the following Python code into properly formatted Google-style function docstrings.
The comments directly following function headers should be moved inside the docstring.
Keep the meaning intact and remove the original inline comments.

CODE:

```
import re
def convert_comments_to_docstrings(code):
    """
    Convert inline comments inside functions into Google-style docstrings.

    Args:
        code (str): Python source code with inline comments.

    Returns:
        str: Python code with comments replaced by structured docstrings.
    """
    lines = code.split('\n')
    new_lines = []
    inside_function = False
    docstring_added = False

    for i, line in enumerate(lines):
        stripped = line.strip()

        # Detect function definition
        if stripped.startswith('def ') and stripped.endswith(':'):
            inside_function = True
            docstring_added = False
            new_lines.append(line)
            continue

        # If inside a function and not yet added docstring
        if inside_function and not docstring_added:
            # Check if the next lines are inline comments (starting with #)
            comment_lines = []
            j = i
            while j < len(lines) and lines[j].strip().startswith('#'):
                comment_lines.append(lines[j].strip()[1:].strip())
                j += 1

            if comment_lines:
```

```
def convert_comments_to_docstrings(code):
                # Create Google-style docstring from comments
                docstring = '    """\n'
                for c in comment_lines:
                    docstring += f"    {c}\n"
                docstring += '    """'
                new_lines.append(docstring)
                docstring_added = True
                # Skip the original comment lines
                for _ in range(len(comment_lines)):
                    next(lines, None)
                inside_function = True
                continue
            else:
                # No comment lines found, just proceed
                new_lines.append(line)
                inside_function = False
                continue

        # Default: just add line
        new_lines.append(line)

    return '\n'.join(new_lines)

# Example usage
if __name__ == "__main__":
    sample_code = '''
def add(a, b):
    # Adds two numbers and returns the result
    return a + b

def greet(name):
    # Prints a greeting message
    print(f"Hello, {name}!")
'''
    converted_code = convert_comments_to_docstrings(sample_code)
    print(converted_code)
```

OUTPUT:

OBSERVATION:

Effective documentation is essential for producing maintainable, readable, and professional-quality code. Tasks focused on improving Python code documentation—whether by adding Google-style docstrings, converting inline comments to structured docstrings, reviewing and correcting outdated docstrings, or comparing the impact of different prompts—highlight the value of clear and consistent documentation.

Task Description #5 (Documentation – Review and Correct Docstrings)

- Task: Use AI to identify and correct inaccuracies in existing docstrings.

	• Instructions:
	  ◦ Provide Python code with outdated or incorrect docstrings.
	  ◦ Instruct AI to rewrite each docstring to match the current code behavior.
	  ◦ Ensure corrections follow Google-style formatting.
	• Expected Output #5:
	  ◦ Python file with updated, accurate, and standardized docstrings.
	PROMPT:
	Please review the following Python code and identify any inaccurate or outdated function docstrings.
	Rewrite each docstring to accurately describe the current behavior of the function.
	Ensure all docstrings follow the Google-style formatting, including sections for description, arguments with type hints, and return values with type hints.
	Do not alter the code logic itself, only improve the docstrings.
	
	CODE:	

```
def correct_docstrings_simple():
    code = '''
def add(a, b):
    """
    Adds two strings (incorrect).
    """
    return a + b

def greet(name):
    """
    Says goodbye (incorrect).
    """
    print(f"Hello, {name}!")
'''
    corrected_docstrings = {
        'add': '''"""
Adds two numbers together.

Args:
    a (int): First number.
    b (int): Second number.

Returns:
    int: Sum of a and b.
"""''',
        'greet': '''"""
Prints a greeting message.

Args:
    name (str): Name of the person.

Returns:
    None
"""'''
    }
    for func, doc in corrected_docstrings.items():
        start = f'def {func}('
```

```
}
for func, doc in corrected_docstrings.items():
    start = f'def {func}('
    idx = code.find(start)
    if idx != -1:
        start_doc = code.find('"""', idx)
        end_doc = code.find('"""', start_doc + 3)
        if start_doc != -1 and end_doc != -1:
            code = code[:start_doc] + doc + code[end_doc+3:]

print(code)
rect_docstrings_simple()
```

OBSERVATION:

Correcting and standardizing docstrings significantly improves code

	documentation quality. Accurate Google-style docstrings enhance clarity, foster better collaboration, and reduce bugs or misuse caused by misunderstood code. This task is essential for maintaining a high-quality, maintainable codebase.	

	Task Description #6 (Documentation – Prompt Comparison Experiment)
	• Task: Compare documentation output from a vague prompt and a detailed prompt for the same Python function.
	• Instructions:
	  o Create two prompts: one simple ("Add comments to this function") and one detailed ("Add Google-style docstrings with parameters, return types, and examples").
	  o Use AI to process the same Python function with both prompts.
	  o Analyze and record differences in quality, accuracy, and completeness.
	• Expected Output #6:
	  o A comparison table showing the results from both prompts with observations.
	PROMPT:
	• Vague prompt:
	Add comments to this function.
	• Detailed prompt:
	Add Google-style docstrings with parameters, return types, and examples to this function.
	CODE:	

```
def vague_prompt_docstring(func_code):
    """
    Simulates AI response to a vague prompt: "Add comments to this function"
    Returns a simple inline comment version.
    """
    return '''
def add(a, b):
    # Adds two numbers
    return a + b
'''

def detailed_prompt_docstring(func_code):
    """
    Simulates AI response to a detailed prompt: "Add Google-style docstrings with parameters, return
    Returns a full Google-style docstring.
    """
    return '''
def add(a: int, b: int) -> int:
    """
    Adds two integers and returns the result.

    Args:
        a (int): First number.
        b (int): Second number.

    Returns:
        int: Sum of a and b.

    Example:
        >>> add(2, 3)
        5
    """
    return a + b
'''

def compare_documentation(func_code):
    vague_doc = vague_prompt_docstring(func_code)
```

```
def detailed_prompt_docstring(func_code):
'''

def compare_documentation(func_code):
    vague_doc = vague_prompt_docstring(func_code)
    detailed_doc = detailed_prompt_docstring(func_code)

    print("=== Vague Prompt Output ===")
    print(vague_doc)

    print("\n=== Detailed Prompt Output ===")
    print(detailed_doc)

    print("\n=== Comparison Table ===")
    print(f"{'Criteria':<20} | {'Vague Prompt':<30} | {'Detailed Prompt':<40}")
    print("-"*95)
    print(f"{'Clarity':<20} | {'Basic comment, less clear':<30} | {'Clear, detailed explanation':<40
    print(f"{'Completeness':<20} | {'Minimal info':<30} | {'Full info including args, returns, examp
    print(f"{'Accuracy':<20} | {'Accurate but brief':<30} | {'Accurate and comprehensive':<40}")
    print(f"{'Format':<20} | {'Inline comments':<30} | {'Google-style docstrings':<40}")
    print(f"{'Usefulness':<20} | {'Limited for large codebases':<30} | {'Better for maintenance and

if __name__ == "__main__":
    sample_function_code = '''
def add(a, b):
    return a + b
'''
    compare_documentation(sample_function_code)
```

OUTPUT:

```
[Running] python -u "c:\Users\PEDDAPELLI ANUSHA\OneDrive\Desktop\Btech.2nd yr\AI\9.1.6.PY"

[Done] exited with code=0 in 0.428 seconds
```

OBSERVATION:

The detailed prompt results in documentation that is much more

	informative, standardized, and useful, especially for teams or projects requiring clear and maintainable code documentation. The vague prompt yields quick but minimal commentary that might suffice for very small scripts or quick notes but lacks depth and formality.	