Lab 9: Documentation Generation - Automatic Documentation and Code Comments

Course: AI Assisted Coding (23CS002PC304)

Assignment: 9.1

Theme: Automatic Documentation and Code Comments

Task 1: Documentation – Function Summary Generation

Task Description

Use AI to generate concise functional summaries for each Python function in a given script. Provide a Python script to the AI and ask it to write a short summary describing the purpose of each function. Ensure summaries are brief and technically accurate, avoiding code implementation details.

Prompt

> "I have a Python script with mathematical utility functions. Please generate a concise, one-line summary for each function describing its purpose. Do not explain how the code works, just what it does."

Input Code provided to AI

```python
def calculate_circle_area(radius):
    if radius < 0:
        return None
    return 3.14159 * radius ** 2

def is_palindrome(text):
    clean_text = text.replace(" ", "").lower()
    return clean_text == clean_text[::-1]

def get_discounted_price(price, discount_percent):
    return price - (price * discount_percent / 100)
```

Output (AI Response)

```python
def calculate_circle_area(radius):
    """Calculates the area of a circle given its radius, returning None for negative inputs."""
    if radius < 0:
        return None
    return 3.14159 * radius ** 2

def is_palindrome(text):
    """Checks if a string is a palindrome, ignoring spaces and case sensitivity."""
    clean_text = text.replace(" ", "").lower()
    return clean_text == clean_text[::-1]

def get_discounted_price(price, discount_percent):
    """Computes the final price of an item after applying a percentage discount."""
    return price - (price * discount_percent / 100)
```

Explanation

For this task, the goal was brevity. I intentionally asked the AI to avoid implementation details (like mentioning the specific value of Pi or string slicing methods). The AI successfully abstracted the *intent* of the functions. For example, instead of saying "reverses string and compares," it simply said "Checks if a string is a palindrome," which is much more useful for a developer quickly scanning the file.

Task 2: Documentation – Logical Explanation for Conditions and Loops

Task Description

Use AI to document the logic behind conditional statements and loops in a Python program. Provide a Python program without comments and instruct the AI to explain only the decision-making logic and loop behavior, skipping basic syntax explanations.

Prompt

> "Add comments to the following Python code that strictly explain the *logic* behind the if-statements and the while-loop. Do not comment on syntax (like 'variable assignment'). Focus on *why* the decision is being made."

Input Code provided to AI

```python
def process_access_request(user_role, attempts):
    status = "denied"

    if user_role == "admin":
        status = "granted"
    elif user_role == "editor":
        if attempts < 3:
            status = "granted"
        else:
            status = "locked"
    else:
        status = "denied"

    while attempts > 0:
        print(f"Processing... {attempts} left")
        attempts -= 1

    return status
```

Output (AI Response)

```python
def process_access_request(user_role, attempts):
    status = "denied"

    # Grant immediate access if the user is an administrator
    if user_role == "admin":
        status = "granted"
    # Editors are granted access only if they haven't exceeded the attempt limit
    elif user_role == "editor":
        if attempts < 3:
            status = "granted"
        else:
            # Lock the account if an editor has failed too many times
            status = "locked"
    # All other roles are denied access by default
    else:
        status = "denied"

    # Simulate a countdown to process the request until attempts reach zero
    while attempts > 0:
        print(f"Processing... {attempts} left")
        attempts -= 1

    return status
```

Explanation

Here, the prompt was specific about ignoring syntax. Beginners often comment i = i + 1 with "increment i," which is useless. The AI followed the instructions well, explaining the *business logic* (e.g., "Grant immediate access if admin," "Lock the account..."). This type of documentation is crucial for maintenance because it tells future developers *why* the rules exist, not just what the code looks like.

Task 3: Documentation – File-Level Overview

Task Description

Use AI to generate a high-level overview describing the functionality of an entire Python file. Provide the complete Python file to the AI and ask it to write a brief overview summarizing the file's purpose and functionality. Place the overview at the top of the file.

Prompt

> "Read the following Python code which handles a simple banking system. Write a high-level docstring to be placed at the very top of the file that summarizes the

module's purpose and key capabilities."

Input Code provided to AI

(Passed a 50-line script containing a BankAccount class with deposit, withdraw, and balance check methods)

Output (AI Response)

```
"""
Banking System Module

This module provides a simple simulation of a bank account system.
It defines the `BankAccount` class, which allows for:
- Creating new accounts with an initial balance.
- Depositing funds safely.
- Withdrawing funds with insufficient balance checks.
- Retrieving current account details.

Usage:
    account = BankAccount("John Doe", 1000)
    account.withdraw(500)
"""

class BankAccount:
    # ... implementation follows ...
```

Explanation

A file-level docstring acts as the "README" for a specific file. The AI identified the core entity (BankAccount) and listed its capabilities in bullet points. It even provided a small usage example, which wasn't explicitly requested but is a hallmark of good AI-generated documentation. This helps other developers understand if this is the file they need without reading every line of code.

Task 4: Documentation – Refine Existing Documentation

Task Description

Use AI to improve the clarity and consistency of existing documentation in Python code. Provide Python code containing basic or unclear comments and ask the AI to rewrite them.

Prompt

> "The comments in this code are vague and poorly written. Rewrite them to be professional, clear, and descriptive without changing the code's functionality."

Input Code provided to AI

```python
def merge_lists(list_a, list_b):
    # make a new empty list
    result = []
    # loop through first list
    for item in list_a:
        # put item in result
        result.append(item)
    # loop through second list
    for item in list_b:
        # check if item exists
        if item not in result:
            # add it
            result.append(item)
    return result
```

Output (AI Response)

```python
def merge_lists(list_a, list_b):
    """Merges two lists into a single list, ensuring no duplicates from the second list."""

    result = []

    # Add all elements from the primary list
    for item in list_a:
        result.append(item)

    # Append elements from the secondary list only if they are not already present
    for item in list_b:
        if item not in result:
            result.append(item)

    return result
```

Explanation

The original comments were redundant (e.g., # make a new empty list). The AI replaced these with higher-level descriptions. Instead of # loop through second list, it wrote # Append elements from the secondary list only if they are not already present. This shifts the focus from "what is the computer doing?" to "what is the algorithm achieving?", making the code significantly easier to read.

Task 5: Documentation – Prompt Detail Impact Study

Task Description

Study the impact of prompt detail on AI-generated documentation quality. Create two prompts: one brief and one detailed, use both to document the same function, and compare the outputs.

Function Used

```
def complex_sort(data, key_func, reverse=False):
    return sorted(data, key=key_func, reverse=reverse)
```

Prompt 1 (Brief)

> "Add documentation to this function."

Output 1 (Brief Prompt)

```
def complex_sort(data, key_func, reverse=False):
    # Sorts the data using the key and reverse flag
    return sorted(data, key=key_func, reverse=reverse)
```

Prompt 2 (Detailed)

> "Generate a Google-style docstring for this function. Include a description of arguments (Args), the return value (Returns), and a brief explanation of what the function does."

Output 2 (Detailed Prompt)

```python
def complex_sort(data, key_func, reverse=False):
    """
    Sorts a dataset based on a custom key function.

    Args:
        data (list): The list of items to be sorted.
        key_func (callable): A function to extract a comparison key from each element.
        reverse (bool, optional): If True, the list is sorted in descending order. Defaults to False.

    Returns:
        list: A new list containing the sorted elements.
    """
    return sorted(data, key=key_func, reverse=reverse)
```

Comparison & Explanation

Feature	Brief Prompt Output	Detailed Prompt Output
Completeness	Very Low. Just a single inline comment.	High. Includes params, types, and return details.
Format	Informal comment.	Structured Docstring (Google Style).
Utility	Minimal. Tells us what we can already see.	High. Explains *how* to use the parameters.

Conclusion: The detailed prompt produced significantly superior documentation. It turns out that AI defaults to the path of least resistance unless explicitly guided to follow a standard structure (like Sphinx or Google style). This demonstrates that "Prompt Engineering" is essential for generating professional-grade API documentation.