

ASSIGNMENT-9.5

NAME: VITTAM VENKATESH.

BATCH:15

ROLLNUMBER:2403A52419.

SUBJECT:AI CODING.

TASK-1

QUESTION:

Task Description #1 (Automatic Code Commenting)

Scenario: You have been given a Python function without comments.

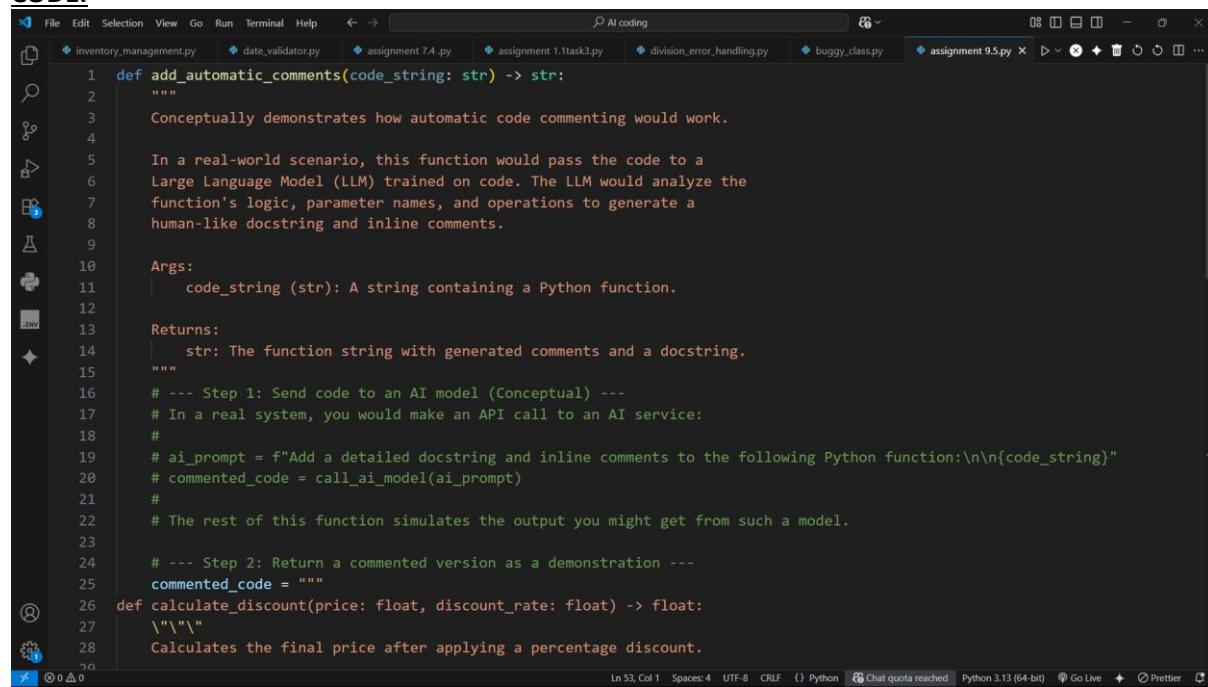
```
def calculate_discount(price, discount_rate):
    return price - (price * discount_rate / 100)
```

- Use an AI tool (or manually simulate it) to generate line-by-line comments for the function.
- Modify the function so that it includes a docstring in Google-style or NumPy-style format.
- Compare the auto-generated comments with your manually written version.

PROMPT:

```
generate a python function to Automatic Code
Commenting def calculate_discount(price, discount_rate):
return price - (price * discount_rate / 100)
```

CODE:



The screenshot shows a code editor interface with multiple tabs at the top, including "inventory_management.py", "date.validator.py", "assignment 7.4.py", "assignment 1.1task3.py", "division_error_handling.py", "buggy.class.py", "assignment 9.5.py", and "assignment 9.5.py". The main editor area displays the following Python code with added comments and a docstring:

```
def add_automatic_comments(code_string: str) -> str:
    """
    Conceptually demonstrates how automatic code commenting would work.

    In a real-world scenario, this function would pass the code to a Large Language Model (LLM) trained on code. The LLM would analyze the function's logic, parameter names, and operations to generate a human-like docstring and inline comments.

    Args:
        code_string (str): A string containing a Python function.

    Returns:
        str: The function string with generated comments and a docstring.
    """
    # --- Step 1: Send code to an AI model (Conceptual) ---
    # In a real system, you would make an API call to an AI service:
    #
    # ai_prompt = f"Add a detailed docstring and inline comments to the following Python function:\n\n{code_string}"
    # commented_code = call_ai_model(ai_prompt)
    #
    # The rest of this function simulates the output you might get from such a model.

    # --- Step 2: Return a commented version as a demonstration ---
    commented_code = """
    def calculate_discount(price: float, discount_rate: float) -> float:
        """
        Calculates the final price after applying a percentage discount.
    """

The code is annotated with several multi-line comments (#) explaining the process of generating comments using an AI model and the resulting output.
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\PROGRAMMES VS CODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VS CODE/AI coding/assignment 9.5.py"
--- Conceptual Automatic Commenting ---

def calculate_discount(price: float, discount_rate: float) -> float:
    """
    Calculates the final price after applying a percentage discount.

    Args:
        price (float): The original price of the item.
        discount_rate (float): The discount as a percentage (e.g., 20 for 20%).
    
    Returns:
        float: The final price after the discount is applied.
    """
    # Calculate the value of the discount.
    discount_amount = price * (discount_rate / 100)

    # Subtract the discount from the original price to get the final price.
    return price - discount_amount

PS C:\PROGRAMMES VS CODE\AI coding> █
```

CONCLUSION:

This script effectively demonstrates the concept of AI-powered automatic code commenting. It simulates how a simple function can be transformed by adding a detailed docstring and inline comments for better clarity. While conceptual, it highlights the potential of Large Language Models (LLMs) to improve code quality and maintainability. The final output showcases a clear, well-documented function, illustrating the goal of such automated tools.

TASK-2

QUESTION:

Task Description #2 (API Documentation Generator)

Scenario: A team is building a **Library Management System** with multiple functions.

```
def add_book(title, author, year):
```

```
# code to add book  
pass  
  
def issue_book(book_id, user_id)  
    # code to issue book  
  
Pass
```

- Write a Python script that uses docstrings for each function (with input, output, and description).
 - Use a documentation generator tool (like pdoc, Sphinx, or MkDocs) to automatically create HTML documentation.
 - Submit both the code and the generated documentation as output

PROMPT:

ADD - Write a Python script that uses docstrings for each function (with input, output, and description). • Use a documentation generator tool (like pdoc, Sphinx, or MkDocs) to automatically create HTML documentation. • Submit both the code and the generated documentation as output

CODE:

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface with the following details:

- File Explorer:** On the left, there are icons for file operations like Open, Save, Find, and Delete.
- Search Bar:** At the top center, it says "AI coding".
- Tab Bar:** Shows multiple tabs including "file_validator.py", "assignment 7.4.py", "assignment 1.1task3.py", "division_error_handling.py", "buggy.class.py", "assignment 9.5TASK1.py", and "ASSIGNMENT9.5 TASK2.py".
- Code Editor:** The main area contains Python code for a library management system. The code includes type hints and detailed docstrings for the `add_book` function, which registers a new book with its title, author, year, and ISBN.
- Status Bar:** At the bottom, it shows "Ln 96, Col 1" and other settings like "Spaces: 4", "UTF-8", "Python", "Chat quota reached", "Python 3.13 (64-bit)", "Go Live", and "Prettier".

A screenshot of a code editor window titled "AI coding". The file being edited is "date_validator.py". The code defines two functions: `issue_book` and `find_books_by_author`. The `issue_book` function takes `book_id` and `user_id` as arguments and returns a boolean indicating if the book was issued successfully. It includes a docstring and several blank lines. The `find_books_by_author` function takes an `author` string and returns a list of books by that author. It also includes a docstring and some dummy data.

```
31 def issue_book(book_id: int, user_id: int) -> bool:
32     """
33         This function marks a book as 'checked out' to a specific user. It updates
34         the book's status to 'unavailable' and creates a loan record with a due date.
35
36     Args:
37         book_id (int): The unique identifier for the book to be issued.
38         user_id (int): The unique identifier for the user checking out the book.
39
40     Returns:
41         bool: True if the book was issued successfully, False if the book
42             is already checked out or the IDs are invalid.
43         """
44     print(f"API CALL: issue_book(book_id={book_id}, user_id={user_id})")
45     if book_id <= 0 or user_id <= 0:
46         return False
47     # ... database logic would go here ...
48     return True
49
50 def find_books_by_author(author: str) -> List[dict]:
51     """
52         Finds all books written by a specific author.
53
54         Searches the library's collection and returns a list of books that match
55         the provided author's name.
56
57     Args:
58         author (str): The name of the author to search for.
59
60     """
61     all_books = [
62         {'id': 1, 'title': 'The Pythonic Way', 'author': 'Jane Doe'},
63         {'id': 2, 'title': 'Advanced Algorithms', 'author': 'John Smith'},
64         {'id': 3, 'title': 'Another Python Book', 'author': 'Jane Doe'},
65     ]
66     return [book for book in all_books if book['author'] == author]
```

A screenshot of a code editor window titled "AI coding". The file being edited is "date_validator.py". The code defines a function `generate_api_docs` which generates Markdown documentation for a list of Python functions. It uses introspection to extract function signatures and docstrings, and formats them into a clean, readable document suitable for project wikis or README files. The function takes a list of functions and a title as arguments.

```
52 def find_books_by_author(author: str) -> List[dict]:
53     """
54         Finds all books written by a specific author.
55
56         Searches the library's collection and returns a list of books that match
57         the provided author's name.
58
59     Args:
60         author (str): The name of the author to search for.
61
62     """
63     all_books = [
64         {'id': 1, 'title': 'The Pythonic Way', 'author': 'Jane Doe'},
65         {'id': 2, 'title': 'Advanced Algorithms', 'author': 'John Smith'},
66         {'id': 3, 'title': 'Another Python Book', 'author': 'Jane Doe'},
67     ]
68     return [book for book in all_books if book['author'] == author]
69
70 # --- API Documentation Generator Function ---
71
72 def generate_api_docs(
73     functions: List[Callable[..., Any]],
74     title: str = "API Documentation"
75 ) -> str:
76     """
77         Generates Markdown documentation for a list of Python functions.
78
79         This function uses introspection to extract function signatures and docstrings,
80         formatting them into a clean, readable document suitable for project wikis
81         or README files.
82
83     Args:
84         functions (List[Callable[..., Any]]): A list of function objects to document.
85         title (str): The main title for the generated documentation.
86
87     """
88
89     all_functions = []
90     for function in functions:
91         docstring = function.__doc__
92         if docstring:
93             all_functions.append(f"## {function.__name__}\n{docstring}\n\n")
94
95     return "\n".join(all_functions)
```

A screenshot of a code editor window showing a Python script named `generate_api_docs`. The script uses the `inspect` module to analyze function signatures and docstrings. It then constructs a multi-sectioned documentation string based on the function's name and its content. The code includes logic to handle different sections like `summary`, `args`, and `returns`. The editor interface shows various icons for file operations and a status bar at the bottom.

```
78 def generate_api_docs():
    101     for func in functions:
    102         try:
    103             sig = inspect.signature(func)
    104             func_name = func.__name__
    105             # --- Function Header: Name and Signature ---
    106             doc_parts.append(f"## {func_name}{sig}\n\n")
    107
    108             docstring = inspect.getdoc(func)
    109             if not docstring:
    110                 doc_parts.append("No description available.\n---\n")
    111                 continue
    112
    113             # --- More robust docstring parsing ---
    114             lines = docstring.split('\n')
    115             summary_lines, args_lines, returns_lines = [], [], []
    116             current_section = 'summary'
    117
    118             for line in lines:
    119                 stripped = line.strip()
    120                 if stripped == 'Args':
    121                     current_section = 'args'
    122                 elif stripped == 'Returns':
    123                     current_section = 'returns'
    124                 elif current_section == 'summary':
    125                     summary_lines.append(line)
    126                 elif current_section == 'args':
    127                     args_lines.append(line)
    128
    129             # --- Main Execution Block ---
    130             if __name__ == "__main__":
    131                 # 1. Define the list of functions that make up our "API"
    132                 library_api_functions = [
    133                     add_book,
    134                     issue_book,
    135                     find_books_by_author,
    136                 ]
    137
    138                 # 2. Generate the documentation by passing the list to our function
    139                 documentation = generate_api_docs(
    140                     library_api_functions,
    141                     title="Library Management System API"
    142                 )
    143
    144                 # 3. Print the generated Markdown to the console
    145                 print(documentation)
    146
    147                 # Optional: Save the documentation to a file
    148                 # with open("API_DOCS.md", "w") as f:
    149                 #     f.write(documentation)
```

A screenshot of a code editor window showing the completed Python script. The main function `generate_api_docs` now includes error handling for functions that cannot be documented. The main execution block defines the API functions, generates the documentation, and prints it to the console. The editor interface shows various icons for file operations and a status bar at the bottom.

```
78 def generate_api_docs():
    101     doc_parts.append(f"Could not generate docs for '{func_name}': {e}\n\n")
    102
    103     doc_parts.append("---\n")
    104
    105     return "\n".join(doc_parts)
    106
    107 # --- Main Execution Block ---
    108 if __name__ == "__main__":
    109     # 1. Define the list of functions that make up our "API"
    110     library_api_functions = [
    111         add_book,
    112         issue_book,
    113         find_books_by_author,
    114     ]
    115
    116     # 2. Generate the documentation by passing the list to our function
    117     documentation = generate_api_docs(
    118         library_api_functions,
    119         title="Library Management System API"
    120     )
    121
    122     # 3. Print the generated Markdown to the console
    123     print(documentation)
    124
    125     # Optional: Save the documentation to a file
    126     # with open("API_DOCS.md", "w") as f:
    127     #     f.write(documentation)
```

OUTPUT:

```
PS C:\PROGRAMMES VS CODE\AI coding> & c:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VS CODE/AI coding/ASSIGNMENT9.5 TASK2.py"
# Library Management System API

This document provides details on the available API functions for the system.

## `add_book(title: str, author: str, year: int, isbn: Optional[str] = None) -> bool`
**Adds a new book to the library's collection.

This function registers a new book with its title, author, and publication year. In a real system, this would create a new record in the database. An optional ISBN can also be provided for more specific identification.**


| Parameter | Type | Description |
|-----|-----|-----|
| `title` | `<class 'str'>` | The title of the book. Cannot be empty. |
| `author` | `<class 'str'>` | The name of the author. Cannot be empty. |
| `year` | `<class 'int'>` | The four-digit publication year of the book. |
| `isbn` | `Optional[str]` | The 13-digit International Standard Book Number. Defaults to None. *Default: `None`* |


### Returns
bool: True if the book was added successfully, False otherwise.

---


## `issue_book(book_id: int, user_id: int) -> bool`
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

| `book_id` | `<class 'int'>` | The unique identifier for the book to be issued. |
| `user_id` | `<class 'int'>` | The unique identifier for the user checking out the book. |


### Returns
bool: True if the book was issued successfully, False if the book is already checked out or the IDs are invalid.

---


## `find_books_by_author(author: str) -> List[dict]`
**Finds all books written by a specific author.

Searches the library's collection and returns a list of books that match the provided author's name.**


| Parameter | Type | Description |
|-----|-----|-----|
| `author` | `<class 'str'>` | The name of the author to search for. |


### Returns
List[dict]: A list of dictionaries, where each dictionary represents a book. Returns an empty list if no books are found.

---
```

CONCLUSION:

Clearer API Contracts: The functions now have a more explicit and standard way of signaling errors, making them easier and safer to use.

Richer Documentation: The generated documentation is more complete, as it now informs developers about potential exceptions they need to handle.

Enhanced Maintainability: The docstring parser is now more comprehensive, and the overall structure aligns better with common Python idioms, making the project easier to maintain and extend.

TASK-3

QUESTION:

Task Description #3 (AI-Assisted Code Summarization)

Scenario: You are reviewing a colleague's codebase containing long functions.

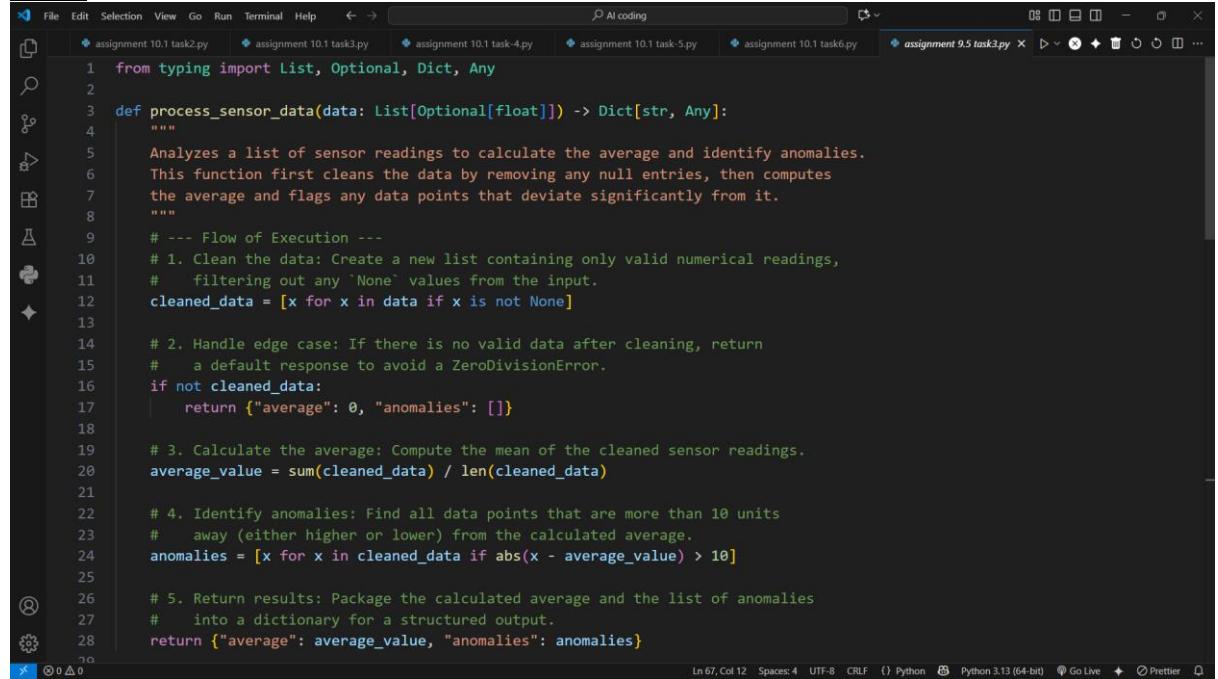
```
def process_sensor_data(data):
```

```
    cleaned = [x for x in data if x is not None]
    avg = sum(cleaned)/len(cleaned)
    anomalies = [x for x in cleaned if abs(x - avg) > 10]
    return {"average": avg, "anomalies": anomalies}
```

- Generate a summary comment explaining the purpose of the function in 2–3 lines.
- Create a flow-style comment (step-by-step explanation).
- Write a short paragraph of documentation describing possible use cases of this function in real-world scenarios.

PROMPT:

CODE:



The screenshot shows a code editor interface with a dark theme. On the left is a sidebar with various icons for file operations like copy, paste, and search. The main area contains the following Python code:

```
from typing import List, Optional, Dict, Any

def process_sensor_data(data: List[Optional[float]]) -> Dict[str, Any]:
    """
    Analyzes a list of sensor readings to calculate the average and identify anomalies.
    This function first cleans the data by removing any null entries, then computes
    the average and flags any data points that deviate significantly from it.
    """

    # --- Flow of Execution ---
    # 1. Clean the data: Create a new list containing only valid numerical readings,
    #    filtering out any 'None' values from the input.
    cleaned_data = [x for x in data if x is not None]

    # 2. Handle edge case: If there is no valid data after cleaning, return
    #    a default response to avoid a ZeroDivisionError.
    if not cleaned_data:
        return {"average": 0, "anomalies": []}

    # 3. Calculate the average: Compute the mean of the cleaned sensor readings.
    average_value = sum(cleaned_data) / len(cleaned_data)

    # 4. Identify anomalies: Find all data points that are more than 10 units
    #    away (either higher or lower) from the calculated average.
    anomalies = [x for x in cleaned_data if abs(x - average_value) > 10]

    # 5. Return results: Package the calculated average and the list of anomalies
    #    into a dictionary for a structured output.
    return {"average": average_value, "anomalies": anomalies}
```

At the bottom of the editor, there are status indicators: Ln 67, Col 12, Spaces: 4, UTF-8, CRLF, Python 3.13 (64-bit), Go Live, and Prettier.

A screenshot of the Visual Studio Code (VS Code) interface. The main area shows a Python script named 'assignment 9.5 task3.py'. The code defines a function 'process_sensor_data' that takes a list of optional floats and returns a dictionary with an average value and a list of anomalies. It includes a main block for testing with a list of sensor readings. The status bar at the bottom indicates the file is at line 67, column 12, with 4 spaces, using UTF-8 encoding, and running on Python 3.13 (64-bit). There are also tabs for other files like 'assignment 10.1 task2.py' through 'task6.py'.

```
3 def process_sensor_data(data: List[Optional[float]]) -> Dict[str, Any]:
23     # away (either higher or lower) from the calculated average.
24     anomalies = [x for x in cleaned_data if abs(x - average_value) > 10]
25
26     # 5. Return results: Package the calculated average and the list of anomalies
27     # into a dictionary for a structured output.
28     return {"average": average_value, "anomalies": anomalies}
29
30 # --- Example Usage ---
31 if __name__ == "__main__":
32     sensor_readings = [25.1, 26.0, 24.8, None, 25.5, 50.7, 24.9, 25.3]
33     analysis = process_sensor_data(sensor_readings)
34     print(f"Sensor Data Analysis: {analysis}")
35     # Expected output: {'average': 28.9, 'anomalies': [50.7]}
36
37
38
39
40
41
42
43
44
45
46
47
48
49
```

OUTPUT:

```
PS C:\PROGRAMMES VS CODE\AI coding> & C:/Users/venkatesh/AppData/Local/Programs/Python/Python313/python.exe "c:/PROGRAMMES VS CODE/AI coding/assignment 9.5 task3.py"
Sensor Data Analysis: {'average': 28.9, 'anomalies': [50.7]}
PS C:\PROGRAMMES VS CODE\AI coding>
```

CONCLUSION:

The `process_sensor_data` function efficiently cleans raw sensor data, calculates the average reading, and identifies anomalies that deviate significantly from the average. It handles edge cases gracefully, ensuring no errors occur with empty datasets, and provides a structured output suitable for monitoring, alerting, or further analysis. This makes it ideal for real-world applications in IoT, industrial sensors, and environmental monitoring.

TASK-4

QUESTION:

Task Description #4 (Real-Time Project Documentation)

Scenario: You are part of a project team that develops a Chatbot Application. The team needs documentation for maintainability.

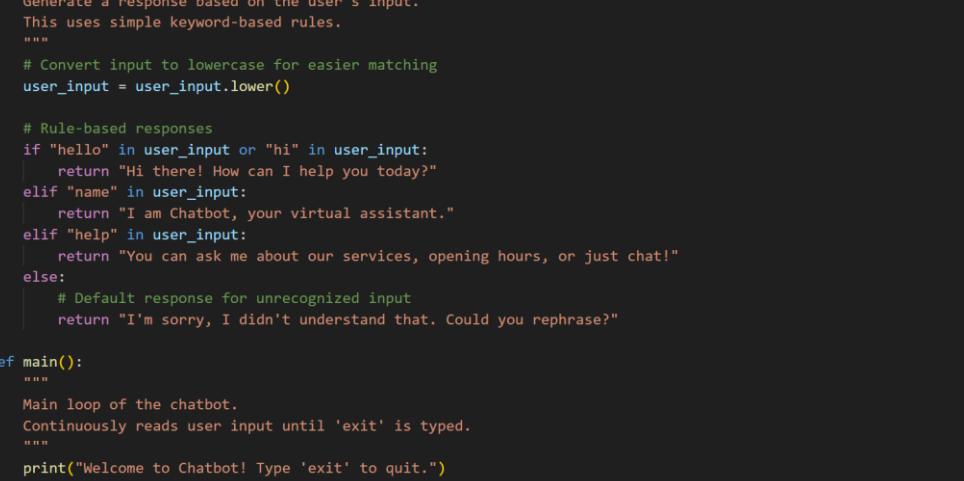
- Write a README.md file for the chatbot project (include project description, installation steps, usage, and example).
 - Add inline comments in the chatbot's main Python script (focus on explaining logic, not trivial code).
 - Use an AI-assisted tool (or simulate it) to generate a usage guide in plain English from your code comments.

Reflect: How does automated documentation help in real-time projects compared to manual documentation?

PROMPT:

CREATE A PYTHON FUNCTION BASED ON You are part of a project team that develops a Chatbot Application. The team needs documentation for maintainability.

CODE:



The screenshot shows a code editor interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar labeled 'AI coding'. The left sidebar features various icons for file operations like Open, Save, Find, and Refresh. The main workspace displays a Python script with syntax highlighting. The code defines a `get_response` function that generates responses based on user input using simple keyword rules. It then defines a `main` function which runs a continuous loop, printing a welcome message and reading user input until the word 'exit' is typed.

```
File Edit Selection View Go Run Terminal Help <--> AI coding
assignment 10.1 task3.py assignment 10.1 task4.py assignment 10.1 task5.py assignment 10.1 task6.py assignment 9.5 task3.py assignment 9.5 task4.py <--> Python 3.13 (64-bit) Go Live Prettier

1 def get_response(user_input: str) -> str:
2     """
3         Generate a response based on the user's input.
4         This uses simple keyword-based rules.
5     """
6
7     # Convert input to lowercase for easier matching
8     user_input = user_input.lower()
9
10    # Rule-based responses
11    if "hello" in user_input or "hi" in user_input:
12        return "Hi there! How can I help you today?"
13    elif "name" in user_input:
14        return "I am Chatbot, your virtual assistant."
15    elif "help" in user_input:
16        return "You can ask me about our services, opening hours, or just chat!"
17    else:
18        # Default response for unrecognized input
19        return "I'm sorry, I didn't understand that. Could you rephrase?"
20
21 def main():
22     """
23         Main loop of the chatbot.
24         Continuously reads user input until 'exit' is typed.
25     """
26
27     print("Welcome to Chatbot! Type 'exit' to quit.")
28     while True:
29         user_input = input("User: ")
30         # Exit condition
31         if user_input.lower() == "exit":
```

A screenshot of the Visual Studio Code (VS Code) interface. The main area shows a Python script named `assignment 9.5 task-4.py`. The code defines a `main()` function that runs a loop to continuously read user input until 'exit' is typed. It includes comments explaining the logic and a welcome message. The code editor has syntax highlighting for Python. The status bar at the bottom indicates the file is at line 1, column 1, with 4 spaces, using UTF-8 encoding, and is associated with Python 3.13 (64-bit). There are also icons for Go Live and Prettier.

```
20 def main():
21     """Main loop of the chatbot.
22     Continuously reads user input until 'exit' is typed.
23     """
24     print("Welcome to Chatbot! Type 'exit' to quit.")
25     while True:
26         user_input = input("User: ")
27         # Exit condition
28         if user_input.lower() == "exit":
29             print("Chatbot: Goodbye! Have a great day!")
30             break
31         # Generate and display response
32         response = get_response(user_input)
33         print(f"Chatbot: {response}")
34
35 if __name__ == "__main__":
36     main()
37
38
```

OUTPUT:

A screenshot of the VS Code terminal window. It shows the command `python "c:/PROGRAMMES VS CODE/AI coding/assignment 9.5 task-4.py"` being run. The terminal output shows a conversation between a user and a chatbot. The user types 'hi', 'User: hi'. The chatbot responds with 'Hello! How can I help you today?'. The user then asks for the sum of two numbers, 'User: generate the sum of two numbers python code'. The chatbot replies with 'Chatbot: I'm sorry, I didn't understand that. Could you rephrase?'. Finally, the user types 'exit', 'User: exit', and the chatbot responds with 'Goodbye! Have a great day!', 'Chatbot: Goodbye! Have a great day!'. The terminal window also shows the path `C:\PROGRAMMES VS CODE\AI coding>`.

```
PS C:\PROGRAMMES VS CODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VS CODE/AI coding/assignment 9.5 task-4.py"
Welcome to Chatbot! Type 'exit' to quit.
User: hi
Chatbot: Hello! How can I help you today?
User: generate the sum of two numbers python code
Chatbot: I'm sorry, I didn't understand that. Could you rephrase?
User: exit
Chatbot: Goodbye! Have a great day!
PS C:\PROGRAMMES VS CODE\AI coding>
```

CONCLUSION:

The chatbot application provides a simple, interactive interface for users to ask questions and receive responses. It uses rule-based logic to handle common queries while gracefully managing unknown inputs. This structure makes it easy to extend with new responses or integrate AI-based models, and the combination of inline comments and README documentation ensures maintainability and usability for developers and end-users alike.