ASSIGNMENT-9.5

NAME: K. Bhuvaneshwar Reddy.

BATCH:15

ROLLNUMBER:2403A52416.

SUBJECT:AI CODING.

TASK-1

QUESTION:

Task Description #1 (Automatic Code Commenting)

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

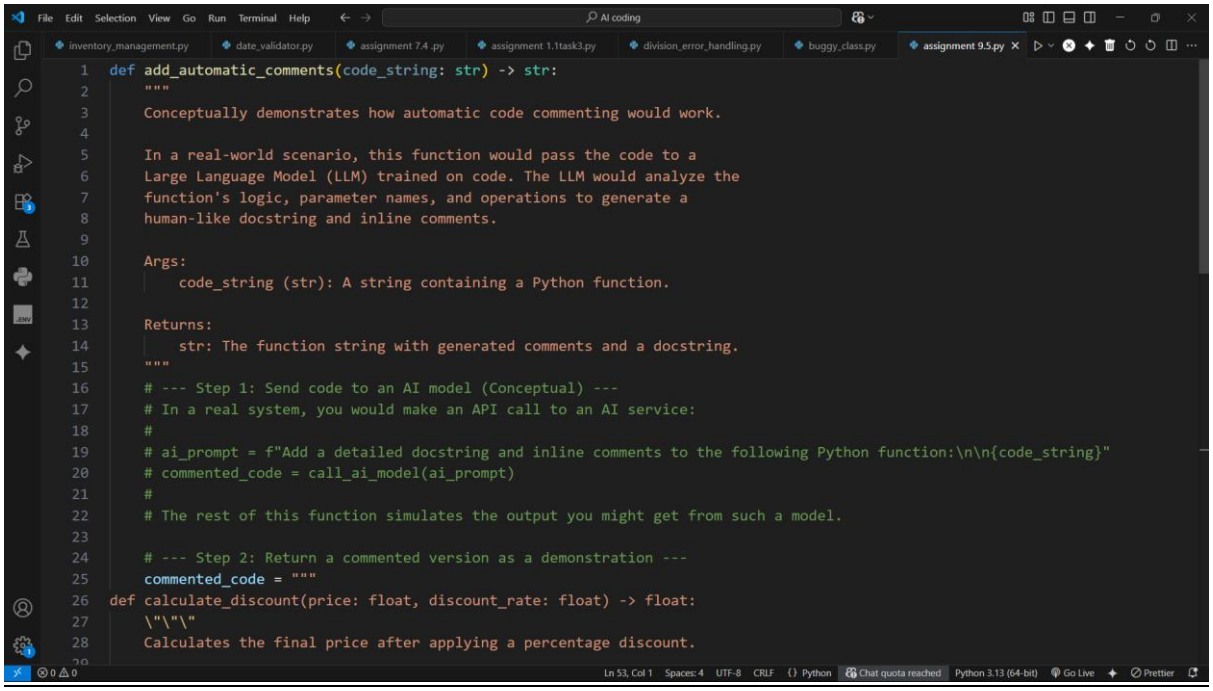
```
def calculate_discount(price, discount_rate):
    return price - (price * discount_rate / 100)
```

- 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.

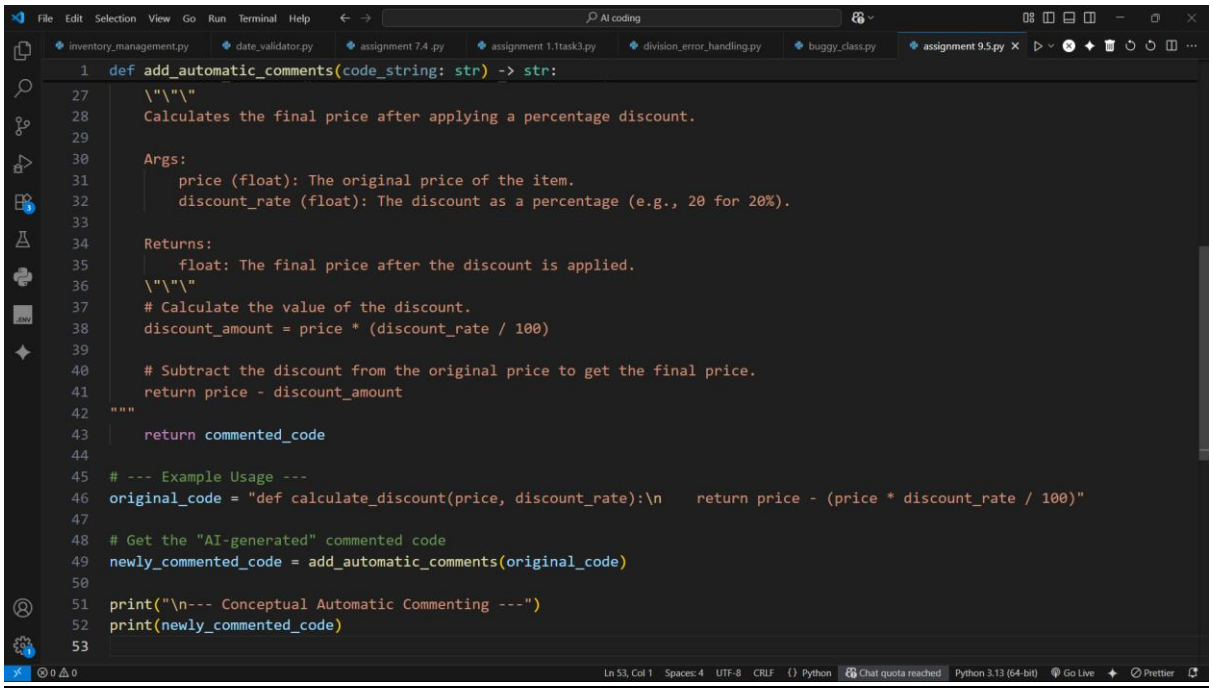
PROMPT:

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

CODE:

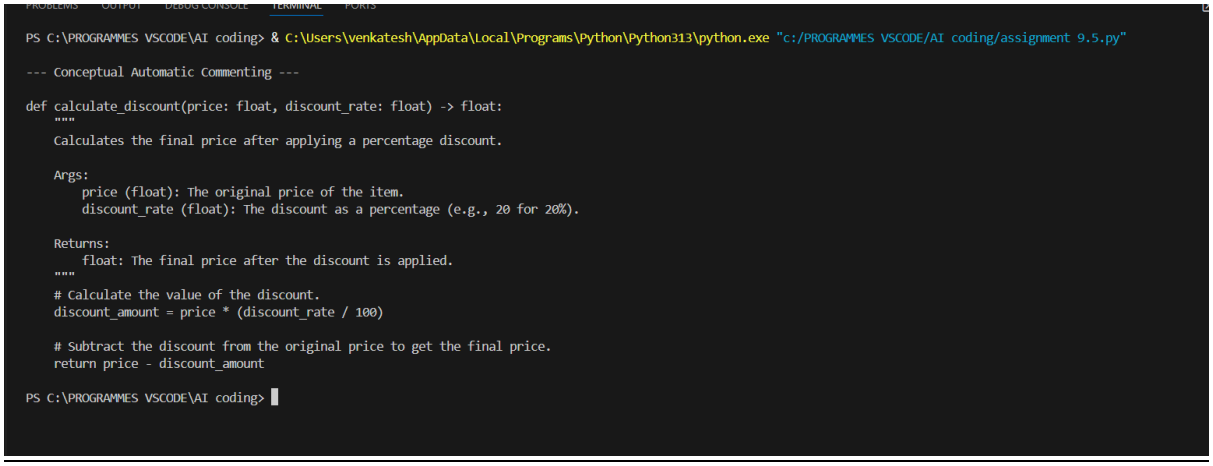


```python
def add_automatic_comments(code_string: str) -> str:
    """
    Conceptually demonstrates how automatic code commenting would work.

    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.
```



```python
def add_automatic_comments(code_string: str) -> str:
    \"\"\"
    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
"""
    return commented_code

# --- Example Usage ---
original_code = "def calculate_discount(price, discount_rate):\n    return price - (price * discount_rate / 100)"

# Get the "AI-generated" commented code
newly_commented_code = add_automatic_comments(original_code)

print("\n--- Conceptual Automatic Commenting ---")
print(newly_commented_code)

```

OUTPUT:



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.

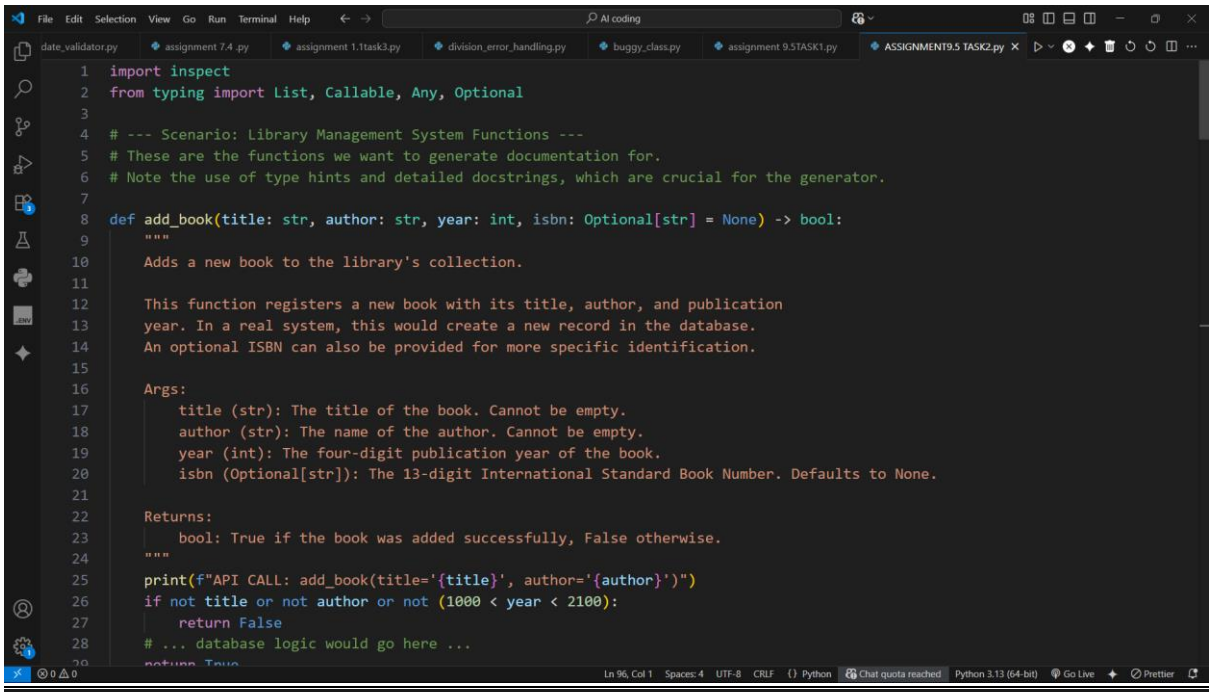
```
def add_book(title, author, year):
    # code to add book
    pass
def issue_book(book_id, user_id):
    # code to issue book
    Pass
```

- 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

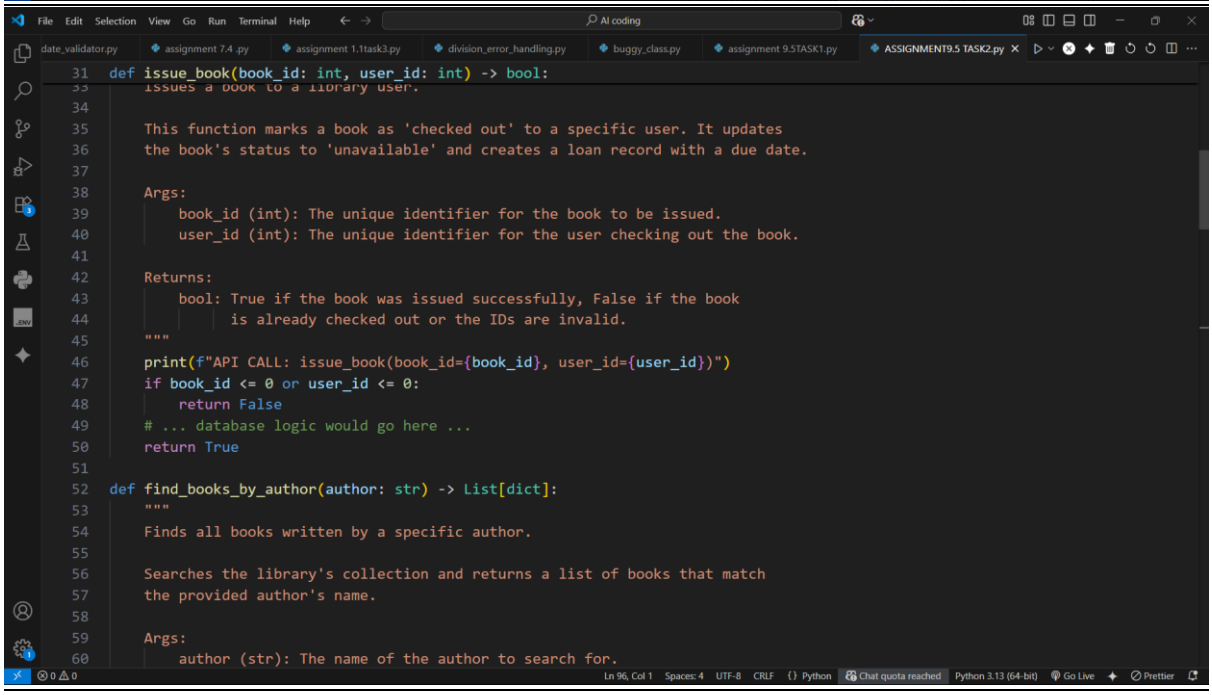
PROMPT:

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

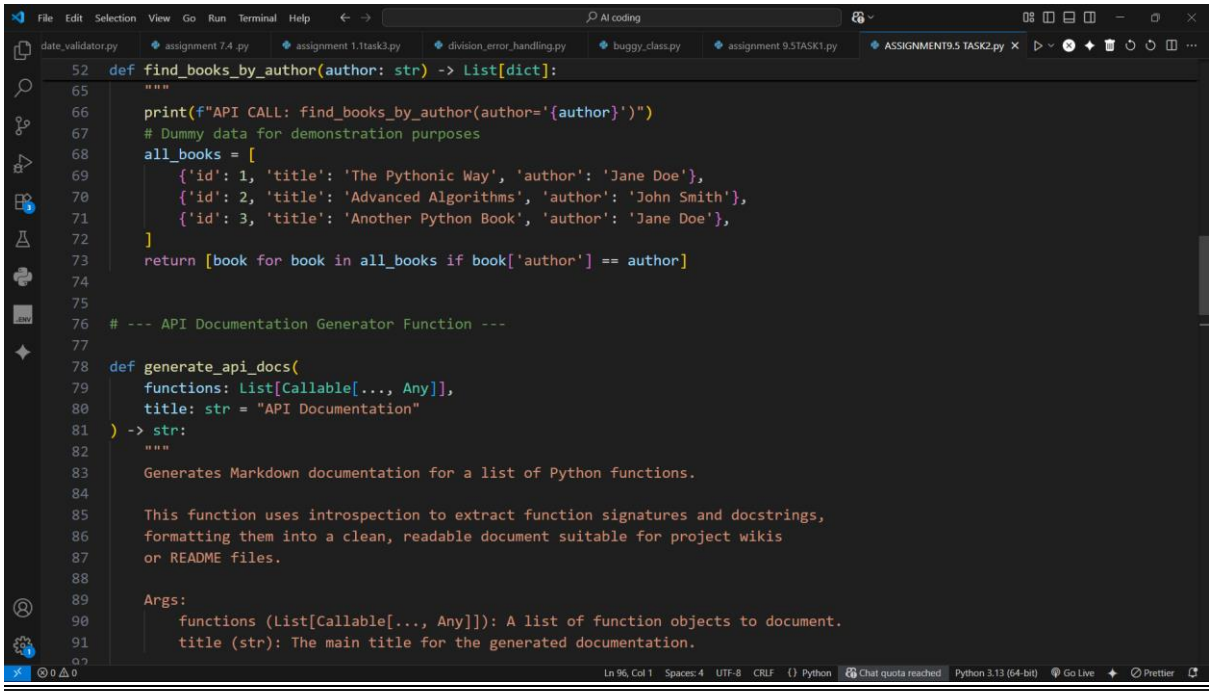
CODE:

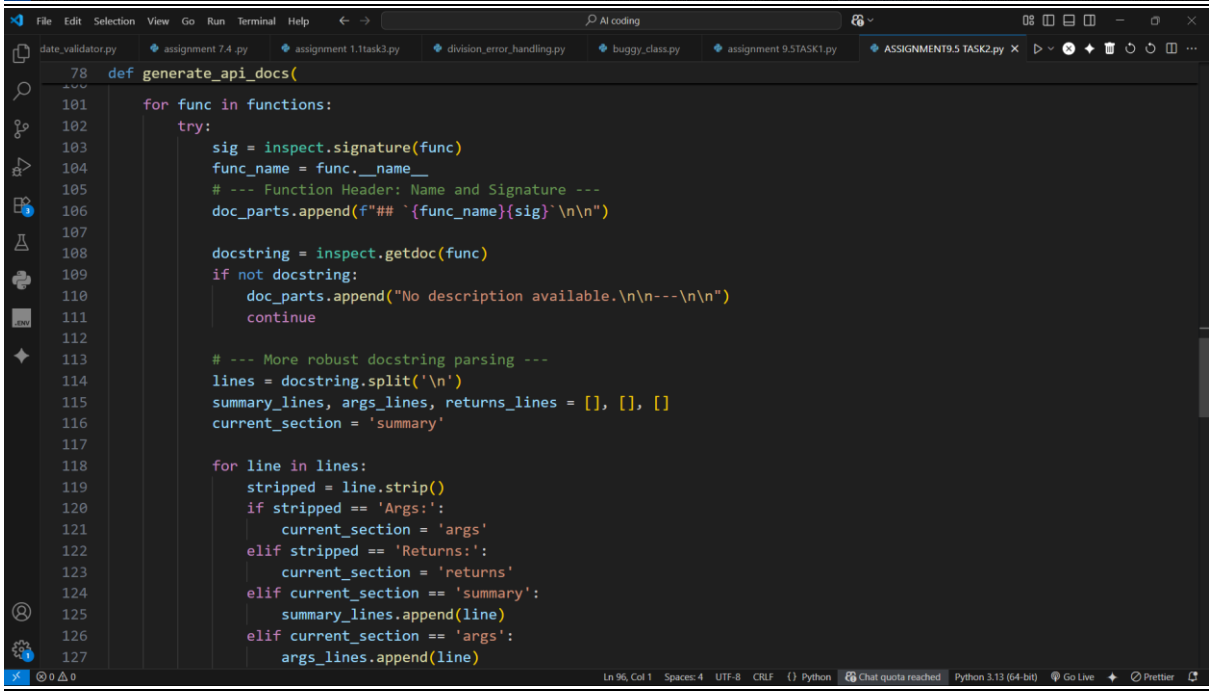


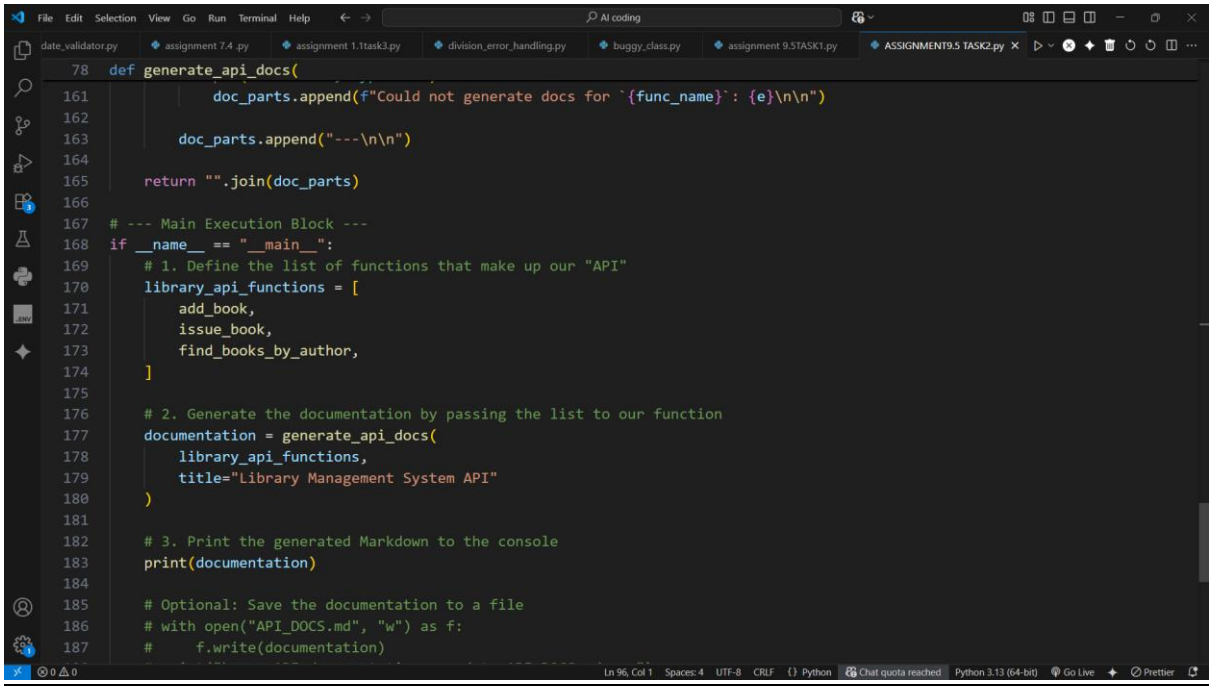
```python
import inspect
from typing import List, Callable, Any, Optional

# --- Scenario: Library Management System Functions ---
# These are the functions we want to generate documentation for.
# Note the use of type hints and detailed docstrings, which are crucial for the generator.

def 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.

    Args:
        title (str): The title of the book. Cannot be empty.
        author (str): The name of the author. Cannot be empty.
        year (int): The four-digit publication year of the book.
        isbn (Optional[str]): The 13-digit International Standard Book Number. Defaults to None.

    Returns:
        bool: True if the book was added successfully, False otherwise.
    """
    print(f"API CALL: add_book(title='{title}', author='{author}')")
    if not title or not author or not (1000 < year < 2100):
        return False
    # ... database logic would go here ...
```



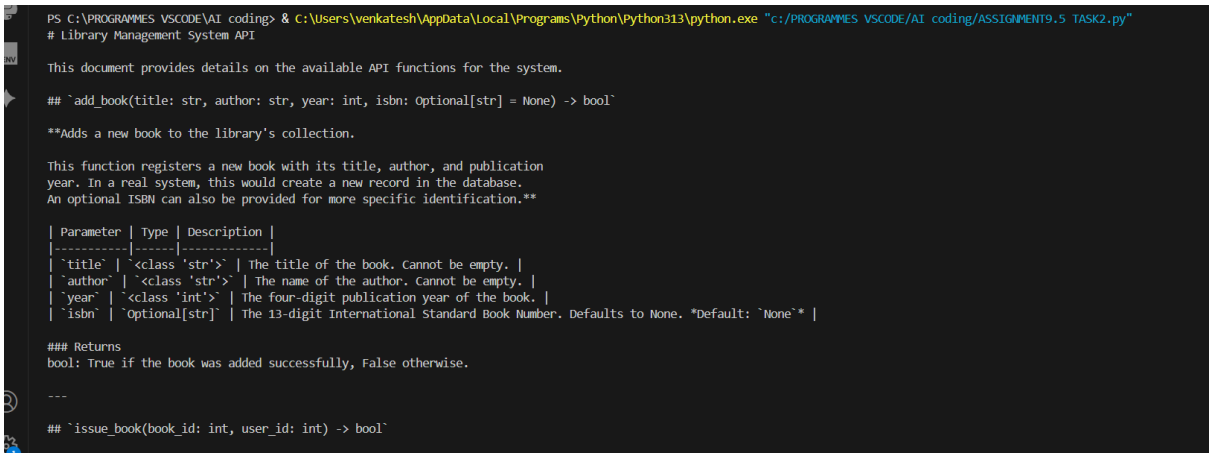
```python
def issue_book(book_id: int, user_id: int) -> bool:
    """
    Issues a book to a library user.

    This function marks a book as 'checked out' to a specific user. It updates
    the book's status to 'unavailable' and creates a loan record with a due date.

    Args:
        book_id (int): The unique identifier for the book to be issued.
        user_id (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.
    """
    print(f"API CALL: issue_book(book_id={book_id}, user_id={user_id})")
    if book_id <= 0 or user_id <= 0:
        return False
    # ... database logic would go here ...
    return True

def 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.

    Args:
        author (str): The name of the author to search for.
```


File Edit Selection View Go Run Terminal Help
AI coding
date_validator.py
assignment 7.4 .py
assignment 1.1task3.py
division_error_handling.py
buggy_class.py
assignment 9.5TASK1.py
ASSIGNMENT9.5 TASK2.py
def find_books_by_author(author: str) -> List[dict]:
"""
print(f"API CALL: find_books_by_author(author='{author}')")
# Dummy data for demonstration purposes
all_books = [
{'id': 1, 'title': 'The Pythonic Way', 'author': 'Jane Doe'},
{'id': 2, 'title': 'Advanced Algorithms', 'author': 'John Smith'},
{'id': 3, 'title': 'Another Python Book', 'author': 'Jane Doe'},
]
return [book for book in all_books if book['author'] == author]
--- API Documentation Generator Function ---
def generate_api_docs(
functions: List[Callable[..., Any]],
title: str = "API Documentation"
) -> str:
"""
Generates Markdown documentation for a list of Python functions.
This function uses introspection to extract function signatures and docstrings,
formatting them into a clean, readable document suitable for project wikis
or README files.
Args:
functions (List[Callable[..., Any]]): A list of function objects to document.
title (str): The main title for the generated documentation.
Ln 96, Col 1
Spaces: 4
UTF-8
CRLF
Python
Chat quota reached
Python 3.13 (64-bit)
Go Live
Prettier

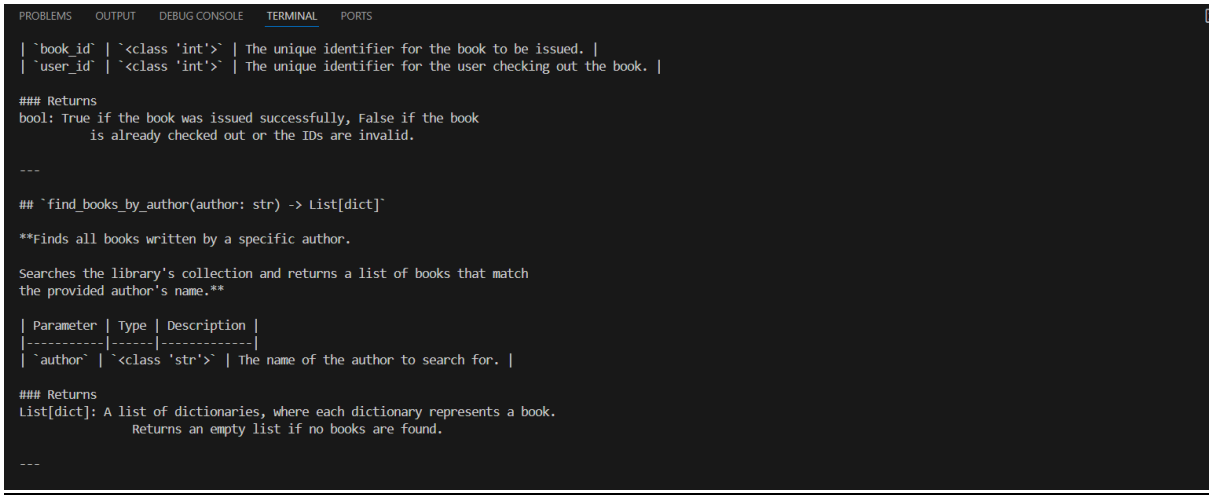

File Edit Selection View Go Run Terminal Help
AI coding
date_validator.py
assignment 7.4 .py
assignment 1.1task3.py
division_error_handling.py
buggy_class.py
assignment 9.5TASK1.py
ASSIGNMENT9.5 TASK2.py
def generate_api_docs(
for func in functions:
try:
sig = inspect.signature(func)
func_name = func.__name__
# --- Function Header: Name and Signature ---
doc_parts.append(f"## `{func_name}{sig}`\n\n")
docstring = inspect.getdoc(func)
if not docstring:
doc_parts.append("No description available.\n\n---\n\n")
continue
# --- More robust docstring parsing ---
lines = docstring.split('\n')
summary_lines, args_lines, returns_lines = [], [], []
current_section = 'summary'
for line in lines:
stripped = line.strip()
if stripped == 'Args:':
current_section = 'args'
elif stripped == 'Returns:':
current_section = 'returns'
elif current_section == 'summary':
summary_lines.append(line)
elif current_section == 'args':
args_lines.append(line)
Ln 96, Col 1
Spaces: 4
UTF-8
CRLF
Python
Chat quota reached
Python 3.13 (64-bit)
Go Live
Prettier


File Edit Selection View Go Run Terminal Help
AI coding
date_validator.py
assignment 7.4 .py
assignment 1.1task3.py
division_error_handling.py
buggy_class.py
assignment 9.5TASK1.py
ASSIGNMENT9.5 TASK2.py
def generate_api_docs(
doc_parts.append(f"Could not generate docs for `{func_name}`: {e}\n\n")
doc_parts.append("---\n\n")
return "".join(doc_parts)
# --- Main Execution Block ---
if __name__ == "__main__":
# 1. Define the list of functions that make up our "API"
library_api_functions = [
add_book,
issue_book,
find_books_by_author,
]
# 2. Generate the documentation by passing the list to our function
documentation = generate_api_docs(
library_api_functions,
title="Library Management System API"
)
# 3. Print the generated Markdown to the console
print(documentation)
# Optional: Save the documentation to a file
# with open("API_DOCS.md", "w") as f:
# f.write(documentation)
Ln 96, Col 1
Spaces: 4
UTF-8
CRLF
Python
Chat quota reached
Python 3.13 (64-bit)
Go Live
Prettier

OUTPUT:


PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/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`



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.

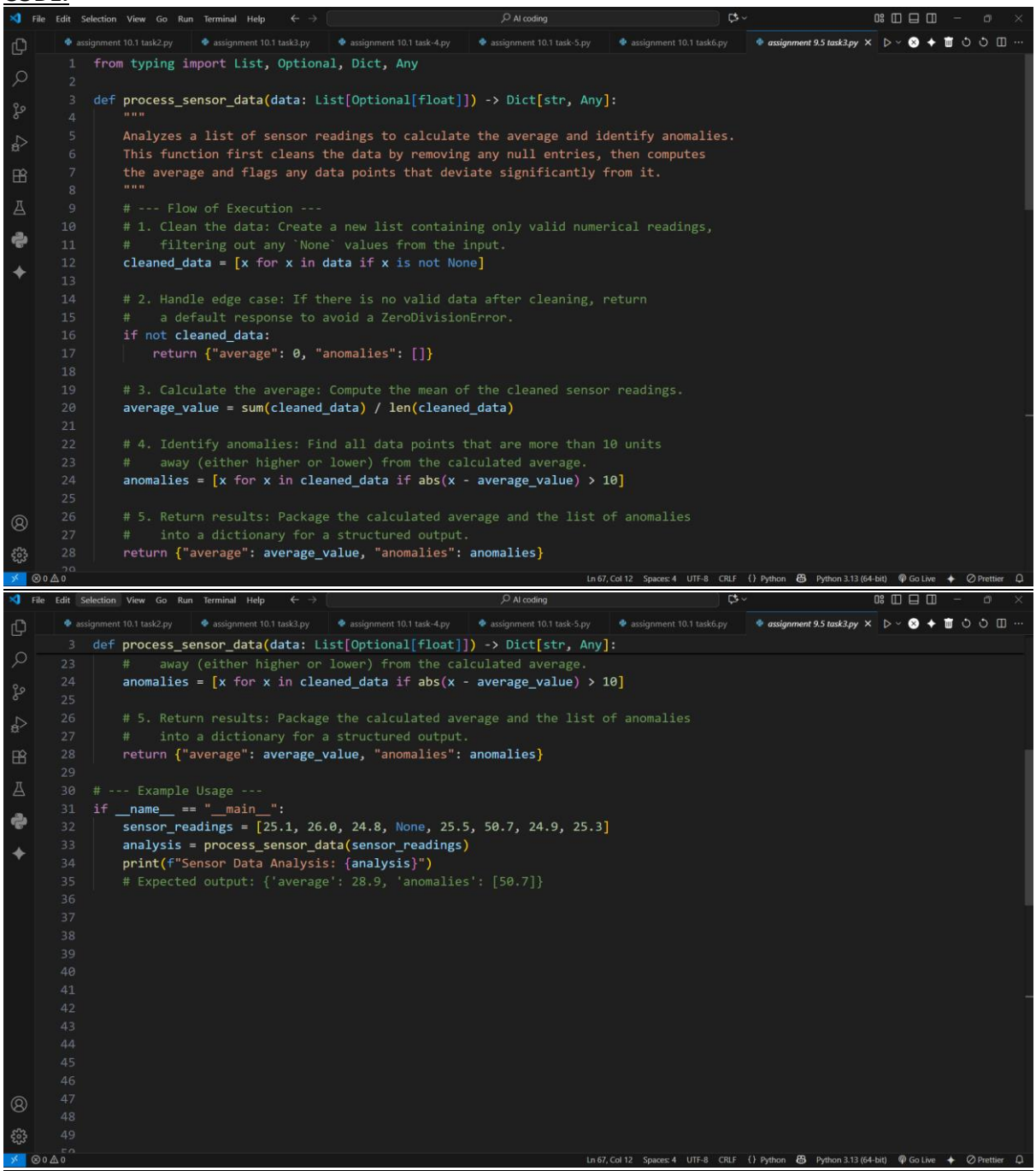
```
def process_sensor_data(data):
    cleaned = [x for x in data if x is not None]
    avg = sum(cleaned)/len(cleaned)
    anomalies = [x for x in cleaned if abs(x - avg) > 10]
    return {"average": avg, "anomalies": anomalies}
```

- 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:

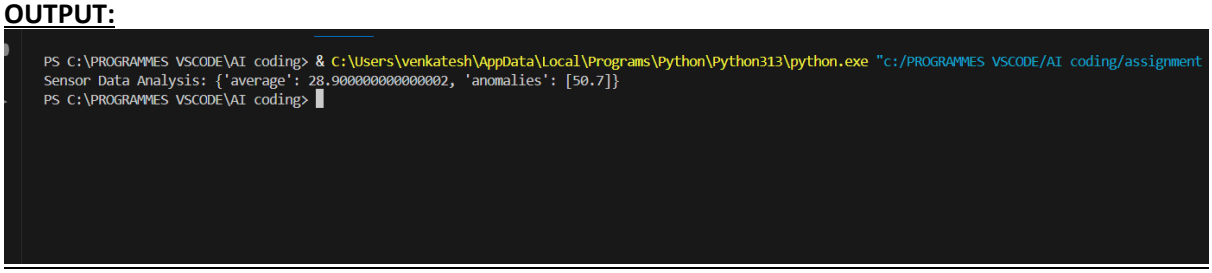


```python
from typing import List, Optional, Dict, Any

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}
```



```python
def process_sensor_data(data: List[Optional[float]]) -> Dict[str, Any]:
    #    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}

# --- Example Usage ---
if __name__ == "__main__":
    sensor_readings = [25.1, 26.0, 24.8, None, 25.5, 50.7, 24.9, 25.3]
    analysis = process_sensor_data(sensor_readings)
    print(f"Sensor Data Analysis: {analysis}")
    # Expected output: {'average': 28.9, 'anomalies': [50.7]}
```

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/assignment
Sensor Data Analysis: {'average': 28.900000000000002, 'anomalies': [50.7]}
PS C:\PROGRAMMES VSCODE\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.

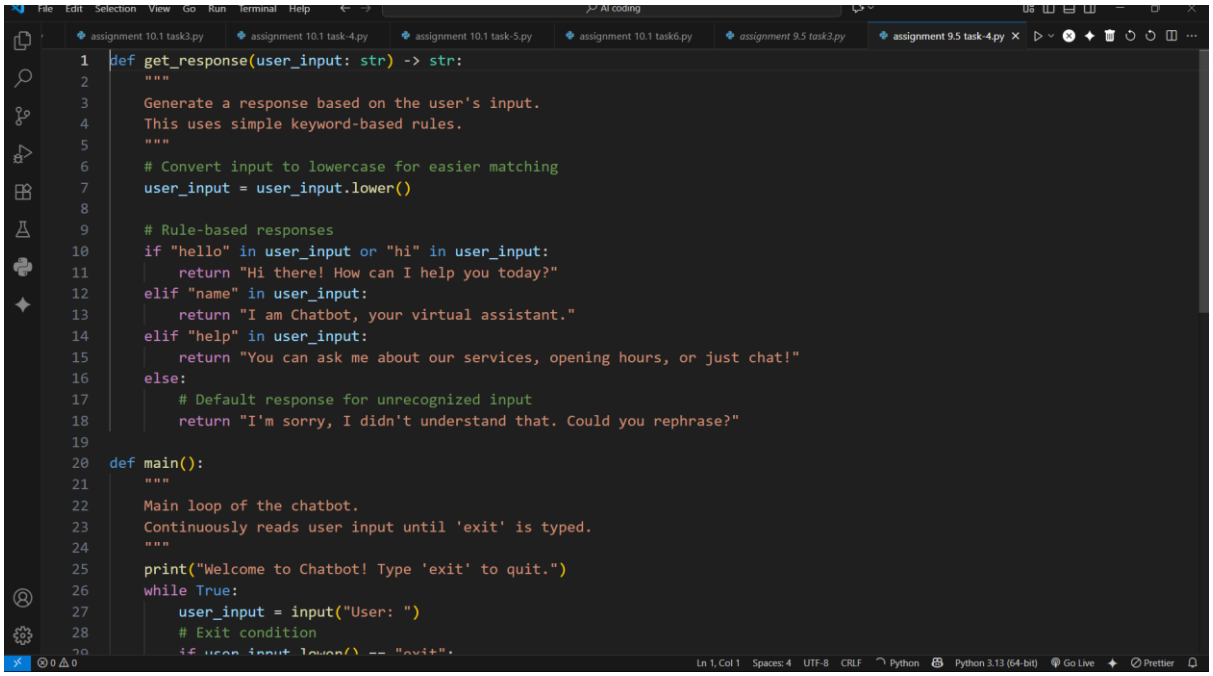
- Write a README.md file for the chatbot project (include project description, installation steps, usage, and example).
- Add inline comments in the chatbot's main Python script (focus on explaining logic, not trivial code).
- Use an AI-assisted tool (or simulate it) to generate a usage guide in plain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

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

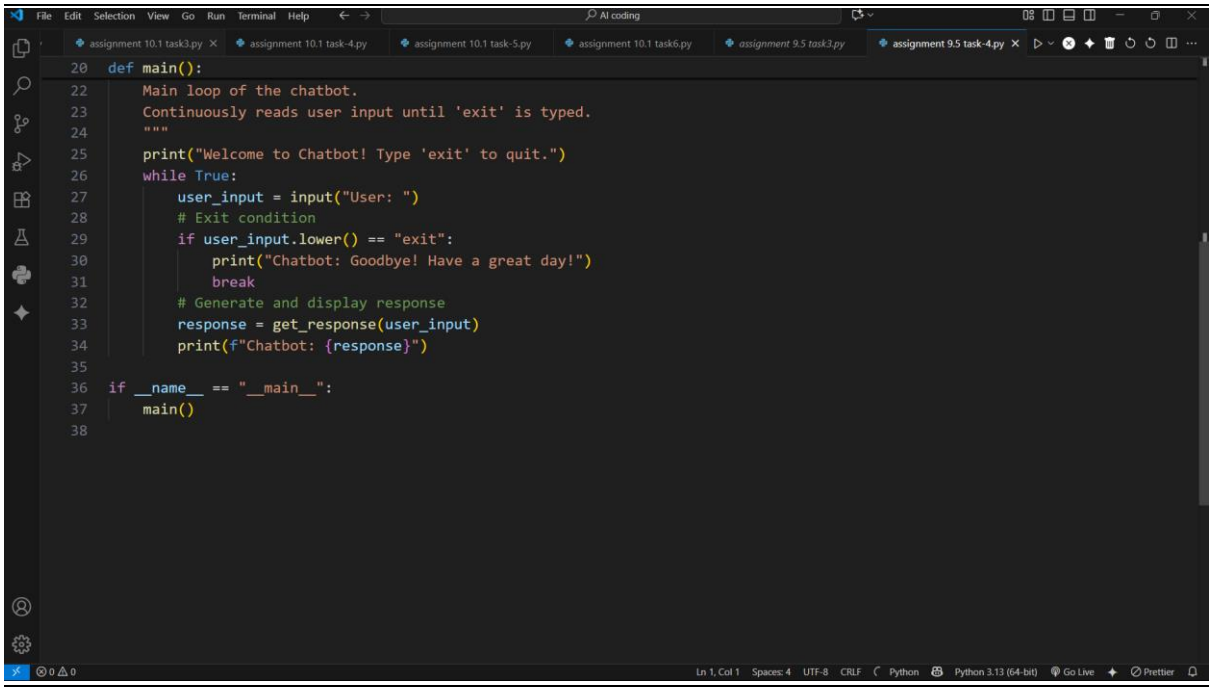
PROMPT:

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:

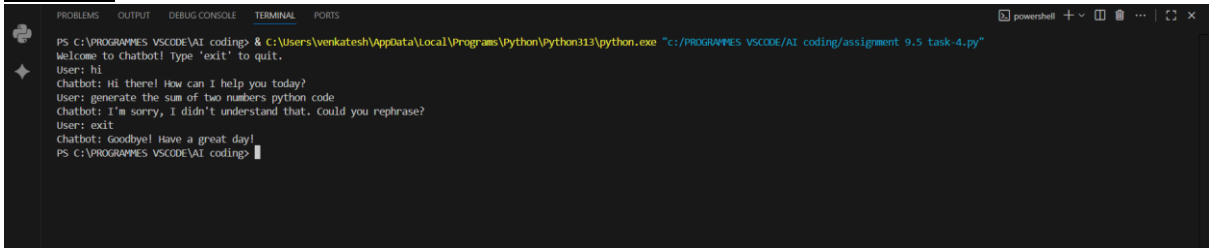


```python
def get_response(user_input: str) -> str:
    """
    Generate a response based on the user's input.
    This uses simple keyword-based rules.
    """
    # Convert input to lowercase for easier matching
    user_input = user_input.lower()

    # Rule-based responses
    if "hello" in user_input or "hi" in user_input:
        return "Hi there! How can I help you today?"
    elif "name" in user_input:
        return "I am Chatbot, your virtual assistant."
    elif "help" in user_input:
        return "You can ask me about our services, opening hours, or just chat!"
    else:
        # Default response for unrecognized input
        return "I'm sorry, I didn't understand that. Could you rephrase?"

def main():
    """
    Main loop of the chatbot.
    Continuously reads user input until 'exit' is typed.
    """
    print("Welcome to Chatbot! Type 'exit' to quit.")
    while True:
        user_input = input("User: ")
        # Exit condition
```



```python
def main():
    """
    Main loop of the chatbot.
    Continuously reads user input until 'exit' is typed.
    """
    print("Welcome to Chatbot! Type 'exit' to quit.")
    while True:
        user_input = input("User: ")
        # Exit condition
        if user_input.lower() == "exit":
            print("Chatbot: Goodbye! Have a great day!")
            break
        # Generate and display response
        response = get_response(user_input)
        print(f"Chatbot: {response}")

if __name__ == "__main__":
    main()
```

OUTPUT:



```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/assignment 9.5 task-4.py"
Welcome to Chatbot! Type 'exit' to quit.
User: hi
Chatbot: Hi there! 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 VSCODE\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.