**HTNO :- 2403A52025**

**Assessment – 9.5**

**Task Description #1 (Automatic Code Commenting)**Scenario: You have been given a Python function without comments.

**CODE**

def calculate\_discount(price, discount\_rate):

"""

Calculate the final price after applying a percentage discount.

Parameters

----------

price : float

The original price of the item.

discount\_rate : float

The discount percentage to apply.

Returns

-------

float

The price after applying the discount.

"""

# price: The original cost of the item before discount

# discount\_rate: The percentage of discount to apply

# (price \* discount\_rate / 100) calculates the discount amount

# Subtract discount amount from the original price

# Return the final payable amount after discount

return price - (price \* discount\_rate / 100)

# Example usage:

original\_price = 1000

discount = 20

final\_price = calculate\_discount(original\_price, discount)

print(f"Original Price: {original\_price}")

print(f"Discount: {discount}%")

print(f"Final Price after discount: {final\_price}")

**OUTPUT**

Original Price: 1000

Discount: 20%

Final Price after discount: 800.0

**OBSERVATION**

 **Functionality**

* Input: price = 1000, discount\_rate = 20.
* Discount amount = 1000 \* 20 / 100 = 200.
* Final Price = 1000 - 200 = 800.0.  
  ✔️ Correct calculation.

 **Auto-generated vs Manual Comments**

* Auto-generated comments are brief (just "apply discount, return value").
* Manual comments explain **parameters, logic, and meaning** more clearly.

 **Docstring**

* Using NumPy-style (or Google-style) makes the function easy to understand when using help(calculate\_discount).
* Example:
* help(calculate\_discount)

would show structured documentation about inputs and outputs.

 **Code Quality**

* With docstrings + manual comments, the function becomes self-explanatory.
* This is better for **team projects, API development, or libraries** where others will reuse your function.

**Task Description #2 (API Documentation Generator)**  
Scenario: A team is building a Library Management System with  
multiple functions.

**CODE**

# library.py

def add\_book(title, author, year):

"""

Add a new book to the library database.

Parameters

----------

title : str

The title of the book.

author : str

The author of the book.

year : int

The publication year of the book.

Returns

-------

dict

A dictionary representing the book added, containing title, author, and year.

"""

# Example implementation (stub)

return {"title": title, "author": author, "year": year}

def issue\_book(book\_id, user\_id):

"""

Issue a book from the library to a specific user.

Parameters

----------

book\_id : int

The unique identifier of the book.

user\_id : int

The unique identifier of the user.

Returns

-------

str

A confirmation message indicating the book has been issued.

"""

# Example implementation (stub)

return f"Book {book\_id} has been issued to User {user\_id}."

**OUTPUT**

{'title': 'The Alchemist', 'author': 'Paulo Coelho', 'year': 1988}

Book 101 has been issued to User 202.

**OBSERVATION**

1. **Code Functionality**
   * add\_book() correctly returns a dictionary with book details.
   * issue\_book() returns a string confirming the book was issued.
2. **Docstrings**
   * Using NumPy-style docstrings makes the documentation structured and professional.
   * Inputs (Parameters) and Outputs (Returns) are clearly described.
3. **Documentation Generator**
   * pdoc automatically reads the docstrings and generates clean HTML pages.
   * The generated docs are easy to navigate and resemble professional API documentation.
4. **Advantages**
   * Keeps **code and documentation in sync** (no need to write docs separately).
   * Useful for **team collaboration** and **project maintainability**.

**Task Description #3 (AI-Assisted Code Summarization)**  
Scenario: You are reviewing a colleague’s codebase containing long  
functions

**CODE**

def process\_sensor\_data(data):

"""

Process raw sensor readings to compute an average and detect simple anomalies.

This function removes `None` entries, computes the arithmetic mean of the remaining

values, and marks any reading whose absolute difference from the mean exceeds 10 units

as an anomaly. It returns a dictionary with keys: "average" and "anomalies".

Notes

-----

- Assumes at least one non-None value is present (otherwise a ZeroDivisionError occurs).

- The anomaly threshold is fixed at 10 units and is not scaled to data variance.

"""

# Flow-style explanation:

# 1) Filter out missing data -> keep only readings that are not None.

cleaned = [x for x in data if x is not None]

# 2) Compute the average of the cleaned readings.

avg = sum(cleaned) / len(cleaned)

# 3) Tag anomalies -> any reading more than 10 units away from the average.

anomalies = [x for x in cleaned if abs(x - avg) > 10]

# 4) Package results in a dictionary for convenient downstream use.

return {"average": avg, "anomalies": anomalies}

**OUTPUT**

{'average': 24.714285714285715, 'anomalies': [50]}

**OBSERVATION**

 **Function Behavior**

* The function:
  + Removes None values → [20, 22, 21, 50, 19, 18, 23]
  + Computes average → (173 / 7) = 24.71
  + Identifies anomalies → values more than ±10 away from average → [50].

 **Code Summarization**

* **Summary comment**: Gives a quick idea of purpose (average + anomaly detection).
* **Flow-style explanation**: Breaks logic into small steps → easy to follow for new developers.
* **Docstring**: Explains assumptions (e.g., at least one valid reading), limitations (fixed threshold), and return format.

 **Use Case Fit**

* Works well for simple anomaly detection in **IoT sensors** (e.g., temp, vibration, water flow).
* However, it’s **basic** → only checks for deviations > 10 units, not statistical anomalies.
* Could fail if:
  + All inputs are None (division by zero).
  + Sensor data varies with large natural fluctuations (false positives).

 **Best Practice**

* Would be improved by:
  + Making threshold configurable.
  + Handling empty/invalid inputs safely.
  + Supporting statistical thresholds (e.g., z-score, standard deviation).

**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.

**CODE**

def get\_response(user\_input):

"""

Generate a chatbot response based on simple keyword matching.

Parameters

----------

user\_input : str

The input message from the user.

Returns

-------

str

The chatbot's reply.

"""

user\_input = user\_input.lower()

# Basic greetings

if "hello" in user\_input or "hi" in user\_input:

return "Hello! How can I help you today?"

# Introduce chatbot

elif "name" in user\_input:

return "I am your friendly chatbot assistant."

# Help response

elif "help" in user\_input:

return "Sure! I can answer basic questions. Try asking me about my name."

# Exit condition handled in main loop

else:

return "I'm not sure how to respond to that."

def main():

print("Chatbot is running! Type 'exit' to quit.")

while True:

# Take input from user

user\_input = input("You: ")

# Check for exit condition

if user\_input.lower() == "exit":

print("Bot: Goodbye!")

break

# Generate response using chatbot logic

response = get\_response(user\_input)

print(f"Bot: {response}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT**

Chatbot is running! Type 'exit' to quit.

You: hi

Bot: Hello! How can I help you today?

You: what is your name?

Bot: I am your friendly chatbot assistant.

You: help

Bot: Sure! I can answer basic questions. Try asking me about my name.

You: something random

Bot: I'm not sure how to respond to that.

You: exit

Bot: Goodbye!

**OBSERVATION**

 **Functionality**

* The chatbot is **rule-based** and works as expected with keyword matching.
* Covers the basic conversational flow (greeting → response → exit).

 **Documentation (README + Inline Comments)**

* The README.md makes it **clear how to install, run, and use** the chatbot.
* Inline comments explain **logic, not trivial code**, which makes it easier for maintainers.

 **AI-Assisted Usage Guide**

* Converts inline comments into a **plain-English explanation** of how the chatbot works.
* Useful for **non-technical team members** or quick onboarding.

 **Reflection on Automation**

* Automated documentation tools (e.g., MkDocs, pdoc) can generate **consistent, up-to-date docs** straight from comments and docstrings.
* This avoids the problem of **manual documentation going stale** in fast-changing projects.

 **Maintainability**

* If new intents are added (e.g., weather queries), updating the docstring + comments would **automatically update the docs site**.
* This makes the project **scalable and team-friendly**.