Open in Colab

In [ ]:
```
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
```

In [2]:
```
def calculate_discount(price, discount_rate):
  """
  Calculates the discounted price based on the original price and discount

  Args:
    price: The original price of the item.
    discount_rate: The discount rate as a percentage.

  Returns:
    The price after applying the discount.
  """
  # Calculate the discount amount
  discount_amount = price * discount_rate / 100
  # Subtract the discount amount from the original price
  discounted_price = price - discount_amount
  # Return the discounted price
  return discounted_price
```

In [ ]:
```
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
```

In [3]:
```
def add_book(title, author, year):
  """Adds a new book to the library system.

  Args:
    title: The title of the book.
    author: The author of the book.
    year: The publication year of the book.

  Returns:
```

```
    A message indicating the book was added successfully (or a book ID).
    (This is a placeholder, actual implementation would return something m
  """
  # code to add book
  print(f"Book '{title}' by {author} ({year}) added.")
  pass

def issue_book(book_id, user_id):
  """Issues a book to a user.

  Args:
    book_id: The ID of the book to issue.
    user_id: The ID of the user issuing the book.

  Returns:
    A message indicating the book was issued successfully (or a transactic
    (This is a placeholder, actual implementation would return something m
  """
  # code to issue book
  print(f"Book with ID {book_id} issued to user {user_id}.")
  pass

# Example usage (optional)
# add_book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams", 1979)
# issue_book(123, 456)
```

In [ ]:

```
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.
```

In [4]:

```
def process_sensor_data(data):
  # Summary:
  # This function processes a list of sensor data,
  # calculates the average after cleaning, and identifies anomalies.
  # It returns the average and a list of anomalous data points.

  # Flow-style explanation:
  # 1. Clean the input data by removing None values.
  # 2. Calculate the average of the cleaned data.
  # 3. Identify data points that are more than 10 units away from the aver
  # 4. Return a dictionary containing the calculated average and the list

  """
  Processes raw sensor data to calculate the average and identify anomalie

  This function is useful in real-world scenarios such as monitoring envir
  conditions (temperature, humidity), analyzing equipment performance data
  or processing financial time series to detect unusual fluctuations.
  It helps in quickly identifying data points that deviate significantly
```

```
  from the norm, which could indicate errors, malfunctions, or significant

  Args:
    data: A list of numerical sensor data, which may contain None values.

  Returns:
    A dictionary containing:
      "average": The average of the cleaned data (excluding None values).
      "anomalies": A list of data points identified as anomalies.
  """
  cleaned = [x for x in data if x is not None]
  if not cleaned:
      return {"average": None, "anomalies": []} # Handle case with no vali
  avg = sum(cleaned)/len(cleaned)
  anomalies = [x for x in cleaned if abs(x - avg) > 10]
  return {"average": avg, "anomalies": anomalies}

# Example usage:
# sensor_readings = [22.5, 23.1, None, 22.8, 35.0, 21.9, 22.7, -5.0]
# result = process_sensor_data(sensor_readings)
# print(result)
```

In [ ]:

```
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?
```

README.md Content Example

In [5]:

```
   python chatbot.py
```

```
  File "/tmp/ipython-input-1300181292.py", line 1
    python chatbot.py
           ^
SyntaxError: invalid syntax
```

In [6]:

```
# chatbot.py - Main script for the simple chatbot

def get_response(user_input):
  """
  Determines the chatbot's response based on user input.

  Args:
    user_input: The string input from the user.

  Returns:
    A string representing the chatbot's response.
  """
  # Convert user input to lowercase for case-insensitive matching
  user_input = user_input.lower()
```

```
  user_input = user_input.lower()

  # Check for specific keywords and return appropriate responses
  if "hello" in user_input or "hi" in user_input:
    # Respond to greetings
    return "Hi there!"
  elif "how are you" in user_input:
    # Respond to inquiries about well-being
    return "I'm a chatbot, I don't have feelings, but I'm ready to help!"
  elif "what is your name" in user_input:
    # Respond to questions about identity
    return "I don't have a name."
  elif "bye" in user_input or "goodbye" in user_input or "quit" in user_ir
    # Respond to farewells and exit command
    return "Goodbye!"
  else:
    # Default response for unrecognized input
    return "I'm sorry, I don't understand."

# Main loop to run the chatbot
def run_chatbot():
  """
  Starts and runs the main chatbot interaction loop.
  """
  print("Simple Chatbot: Hello! Type 'quit' to exit.")

  # Loop indefinitely until the user types 'quit'
  while True:
    # Get input from the user
    user_input = input("You: ")

    # Check if the user wants to quit
    if user_input.lower() == 'quit':
      break

    # Get the chatbot's response
    chatbot_response = get_response(user_input)

    # Print the chatbot's response
    print(f"Chatbot: {chatbot_response}")

# Start the chatbot when the script is executed
if __name__ == "__main__":
  run_chatbot()
```

```
Simple Chatbot: Hello! Type 'quit' to exit.
You: quit
```

Explanation and Reflection

The code above provides a simple example of a chatbot script with inline comments explaining the logic within the `get_response` function. A more complex chatbot would have more intricate logic and potentially separate modules, each requiring detailed comments and docstrings.

Regarding the reflection on automated vs. manual documentation in real-time projects:

Automated Documentation:

Automated Documentation:

- **Pros:**
  - **Consistency:** Ensures documentation follows a consistent format and style (based on the tool's configuration).
  - **Up-to-dateness:** Easily generated from the latest code, reducing the risk of outdated documentation.
  - **Efficiency:** Saves significant time and effort compared to manually writing and updating documentation.
  - **Discoverability:** Can automatically generate indices, cross-references, and search capabilities.
  - **Integration:** Can be integrated into CI/CD pipelines to ensure documentation is always generated with the latest code.
- **Cons:**
  - **Dependency on Code Comments/Docstrings:** Quality of documentation heavily relies on the quality and completeness of comments and docstrings in the code.
  - **Limited Scope:** Primarily focuses on API documentation (functions, classes, methods); may not cover high-level design decisions or architecture unless specifically included in docstrings or separate files processed by the tool.
  - **Setup Overhead:** Initial setup and configuration of the documentation tool can require some effort.

Manual Documentation:

- **Pros:**
  - **Flexibility:** Allows for more narrative explanations, architectural overviews, and design discussions that may not fit well into code comments.
  - **Tailored Content:** Can be specifically written for different audiences (developers, users, managers).
  - **Covers Non-Code Aspects:** Can document project setup, infrastructure, and other aspects not directly in the codebase.
- **Cons:**