Phase-3 DEVELOPMENT PART-1

DATA LOADING:

***Abstraction**:*

Abstraction refers to hiding complex details and showing only the necessary features of an object or system. In the context of data loading for a chatbot, this means creating a clear separation of concerns between data storage and data retrieval. You should abstract the data access methods so that the chatbot's core logic is decoupled from the underlying data source.

***Modulation**:*

Modulation, in the context of software design, involves breaking down the code into smaller, manageable modules or components. Each module should have a specific responsibility and should be loosely coupled with other modules. For data loading, you might have a separate module responsible for retrieving data from various sources and transforming it into a format that the chatbot can use.

Here's an example of how you can design the data loading process for a chatbot:

CODE:

```
# chatbot_data_loader.py


class ChatbotDataLoader:
    def __init__(self, data_source):
        self.data_source = data_source


    def load_data(self):
        # Implement data loading logic based on the data source
        if self.data_source == "database":
            return self.load_data_from_database()
        elif self.data_source == "file":
            return self.load_data_from_file()
```

```
        else:
            raise ValueError("Invalid data source")

    def load_data_from_database(self):
        # Implement code to fetch data from a database
        # Example:
        # return database.query("SELECT question, answer FROM chat_data")

    def load_data_from_file(self):
        # Implement code to load data from a file (e.g., JSON, CSV)
        # Example:
        # with open("chat_data.json", "r") as file:
        #     return json.load(file)

# chatbot.py

class Chatbot:
    def __init__(self, data_loader):
        self.data_loader = data_loader
        self.data = self.data_loader.load_data()

    def generate_response(self, user_input):
        # Implement chatbot logic here using the loaded data
        pass

# main.py

if __name__ == "__main__":
    data_source = "file"  # Change this to "database" or other sources as needed
    data_loader = ChatbotDataLoader(data_source)
    chatbot = Chatbot(data_loader)
```

```
# Now you can use the chatbot to generate responses based on user input
user_input = input("User: ")
response = chatbot.generate_response(user_input)
print("Chatbot:", response)
```

DATA MODELING:

Abstraction:

Data cleaning for a chatbot involves the following steps:

1. **Text Preprocessing:** Removing or transforming noisy and irrelevant information such as special characters, extra whitespaces, and HTML tags.

2. **Tokenization:** Breaking down sentences into individual words or tokens for easier analysis.

3. **Stopword Removal:** Eliminating common words that don't carry much meaning, such as "the," "and," etc.

4. **Stemming or Lemmatization:** Reducing words to their root form to capture the core meaning.

5. **Data Structuring:** Organizing the data in a way that the chatbot can access and utilize it effectively.

CODE:

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Sample user input
user_input = "How can I learn Python programming more effectively?"
```

```
# Step 1: Text Preprocessing
def preprocess_text(text):
    # Remove special characters and convert to lowercase
    text = text.lower()
    text = ''.join([char for char in text if char.isalnum() or char.isspace()])

    # Remove extra whitespaces
    text = ' '.join(text.split())

    return text

cleaned_input = preprocess_text(user_input)

# Step 2: Tokenization
tokens = word_tokenize(cleaned_input)

# Step 3: Stopword Removal
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word not in stop_words]

# Step 4: Stemming or Lemmatization
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]

# Step 5: Data Structuring (Here, we'll simply join the tokens back into a string)
cleaned_input = ' '.join(stemmed_tokens)

# Output the cleaned input
print("Cleaned Input:", cleaned_input)
```

OUTPUT:

Cleaned Input: learn python program effect

TOKENIZATION:

Tokenization is a crucial step in natural language processing, particularly when creating a chatbot in Python. Tokenization involves breaking down a text into individual units, usually words or subwords, to analyze and process the text effectively. In this example, we'll use the NLTK library for tokenization. Before starting, make sure you have NLTK installed, or you can install it using `pip`:

```bash
pip install nltk
```

Here's a Python code example to demonstrate the abstraction and modulation of tokenization for creating a simple chatbot:

CODE:

```
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')  # Download the necessary data for NLTK (if not already done)

# Abstraction: Tokenization Function
def tokenize(text):
    return word_tokenize(text)

# Modulation: Simple Chatbot
def simple_chatbot():
    print("Simple Chatbot: Hello! How can I assist you today?")

    while True:
```

```
        user_input = input("You: ")

        if user_input.lower() == 'exit':
            print("Simple Chatbot: Goodbye!")
            break

        # Tokenization
        tokens = tokenize(user_input)

        # Respond to user input (modulation)
        response = generate_response(tokens)
        print("Simple Chatbot:", response)

# Example of a simple response generation (modulation)
def generate_response(tokens):
    if 'hello' in tokens:
        return "Hello there!"
    elif 'how' in tokens and 'are' in tokens and 'you' in tokens:
        return "I'm just a computer program, but I'm here to help!"
    else:
        return "I'm not sure how to respond to that."

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

In this code:

- The `tokenize` function abstracts the tokenization process, which utilizes the NLTK library's `word_tokenize` function.

- We create a simple chatbot that continually takes user input and responds.

- The `generate_response` function provides a basic response based on the tokens extracted from the user's input.

Here's a sample interaction with the chatbot:

```
Simple Chatbot: Hello! How can I assist you today?

You: Hi there!

Simple Chatbot: Hello there!

You: How are you doing?

Simple Chatbot: I'm just a computer program, but I'm here to help!

You: What's the weather like today?

Simple Chatbot: I'm not sure how to respond to that.

You: Exit

Simple Chatbot: Goodbye!
```

TEXT NORMALIZATION:

****Abstraction of Text Normalization:****

Text normalization is the process of converting input text into a consistent and simplified format to facilitate natural language processing. This involves several steps, including but not limited to:

1. **Lowercasing: Converting all text to lowercase to ensure uniformity.**

2. **Removing Punctuation: Eliminating punctuation marks, making text easier to analyze.**

3. **Expanding Contractions: Expanding contractions like "don't" to "do not" for better understanding.**

4. **Handling Numbers: Standardizing how numbers are represented (e.g., "5" becomes "five").**

5. **Date Normalization: Recognizing and normalizing date formats (e.g., "02/15/2023" to "February 15, 2023").**

6. **Special Character Handling: Dealing with special characters or symbols as needed.**

7. **Stop Words Removal: Removing common words like "the," "and," "is," which don't carry significant meaning.**

8. **Stemming or Lemmatization: Reducing words to their base form (e.g., "running" to "run").**

****Python Code for Text Normalization:****

CODE:

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
from nltk.tokenize import word_tokenize
import re
nltk.download('punkt')
nltk.download('stopwords')

# Text normalization function
def text_normalization(text):
```

```
    # Lowercasing
    text = text.lower()

    # Removing punctuation
    text = re.sub(r'[^\w\s]', '', text)

    # Expanding contractions (you can create a list of contractions and their expansions)
    contractions = {"don't": "do not", "can't": "cannot"}  # Add more as needed
    words = text.split()
    words = [contractions[word] if word in contractions else word for word in words]
    text = ' '.join(words)

    # Tokenization
    words = word_tokenize(text)

    # Removing stop words
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]

    # Stemming (you can also use lemmatization)
    stemmer = SnowballStemmer('english')
    words = [stemmer.stem(word) for word in words]

    # Rejoin the words to form normalized text
    normalized_text = ' '.join(words)

    return normalized_text

# Example usage
```

input_text = "Don't stop thinking about tomorrow! It's 5 o'clock."

normalized_text = text_normalization(input_text)

print(normalized_text)

```

OUTPUT:

```

not stop think tomorrow 5 oclock

```

ENCODING:

 # Abstraction:

A chatbot typically consists of three key components:

1. **Input Processing:**

- Capture and preprocess user input.

- Use NLP techniques to understand the user's intent and extract entities (such as dates, locations, etc.).

2. **Response Generation:**

- Based on the user's input and the chatbot's training data, generate an appropriate response.

- This can involve machine learning models, rule-based systems, or a combination of both.

3. **Conversation Management:**

- Keep track of the conversation history to maintain context.

- Decide when to switch between different conversation states or modes (e.g., providing information, answering questions, or making small talk).

Code Example:

For this example, we'll use the ChatterBot library, which is a simple Python library for building chatbots. You can install it using pip:

```bash
pip install chatterbot
pip install chatterbot_corpus
```

Here's a basic Python code example:

```python
from chatterbot import ChatBot
from chatterbot.trainers import ChatterBotCorpusTrainer

# Create a new chatbot instance
chatbot = ChatBot('MyBot')

# Create a new trainer for the chatbot
trainer = ChatterBotCorpusTrainer(chatbot)

# Train the chatbot on English language data
trainer.train('chatterbot.corpus.english')

# Start a conversation loop
print("Chatbot: Hello! How can I assist you today?")
```

```
while True:
    user_input = input("You: ")
    if user_input.lower() == 'exit':
        print("Chatbot: Goodbye!")
        break
    response = chatbot.get_response(user_input)
    print("Chatbot:", response)
```

Output:

When you run the code, the chatbot will start a conversation loop:

```
Chatbot: Hello! How can I assist you today?
You: What's the weather like today?
Chatbot: In which city would you like to know the weather?
You: New York
Chatbot: I'm sorry, I don't have access to real-time weather data.
You: Tell me a joke
Chatbot: Why did the scarecrow win an award? Because he was outstanding in his field!
You: exit
Chatbot: Goodbye!
```

DATA SERIALIZATION:

Abstraction and modulation in this context involve designing a modular and abstracted data serialization system that allows the chatbot to store and retrieve

information efficiently. Below, I'll provide a Python example of a simple chatbot and how to use data serialization to store its data.

```python
import pickle

class ChatBot:
    def __init__(self):
        self.data = {'greetings': ['Hello!', 'Hi there!', 'Hey!'],
                     'farewells': ['Goodbye!', 'See you later!', 'Take care!']}

    def respond(self, user_input):
        if user_input.lower() in self.data['greetings']:
            return 'Hello! How can I help you?'
        elif user_input.lower() in self.data['farewells']:
            return 'Goodbye! Have a great day.'
        else:
            return "I'm sorry, I don't understand."

    def add_data(self, category, new_data):
        if category in self.data:
            self.data[category].append(new_data)
        else:
            self.data[category] = [new_data]

    def save_data(self, filename):
        with open(filename, 'wb') as file:
            pickle.dump(self.data, file)

    def load_data(self, filename):
```

```
        with open(filename, 'rb') as file:
            self.data = pickle.load(file)

# Initialize the chatbot
chatbot = ChatBot()

# Interaction with the chatbot
while True:
    print("Type 'exit' to quit.")
    user_input = input("You: ")
    if user_input == 'exit':
        chatbot.save_data('chatbot_data.pkl')
        break
    response = chatbot.respond(user_input)
    print("ChatBot:", response)

# Modifying chatbot data
chatbot.add_data('greetings', 'Hola!')
chatbot.add_data('farewells', 'Adios!')

# Saving and loading data using data serialization
chatbot.save_data('chatbot_data.pkl')

# Creating a new chatbot instance and loading the data
new_chatbot = ChatBot()
new_chatbot.load_data('chatbot_data.pkl')

# Testing the new chatbot's responses
print("Testing the new chatbot:")
```

```
user_input = "Hi there!"

response = new_chatbot.respond(user_input)

print("You:", user_input)

print("ChatBot:", response)
```

DATA AUGUMENTATION:

Abstraction:

1. **Data Collection**: Collect a dataset of conversation samples. You can find existing chatbot datasets or create your own.

2. **Preprocessing**: Preprocess the dataset by tokenizing, lowercasing, and removing stopwords and punctuation.

3. **Data Augmentation**: Apply data augmentation techniques to expand your dataset. This helps the chatbot generalize better.

4. **Modeling**: Train a chatbot model using machine learning or deep learning techniques. You can use libraries like TensorFlow, PyTorch, or specialized chatbot frameworks like Rasa.

5. **User Interface**: Create a user interface to interact with your chatbot. You can use Python libraries like Flask or Django for this.

6. **Testing and Iteration**: Test your chatbot, gather user feedback, and iteratively improve both the data augmentation techniques and the chatbot model.

Modulation (Python code and output):

Here's a simple example of data augmentation using synonyms with NLTK and a basic chatbot using the `nltk.chat` module:

```CODE:
import nltk
from nltk.chat.util import Chat, reflections
from nltk.corpus import wordnet

# Define a function to augment text with synonyms
def augment_with_synonyms(text):
    words = text.split()
    augmented_text = []
    for word in words:
        synonyms = wordnet.synsets(word)
        if synonyms:
            synonym = synonyms[0].lemmas()[0].name()
            augmented_text.append(synonym)
        else:
            augmented_text.append(word)
    return ' '.join(augmented_text)

# Define a list of reflections for the chatbot
reflections = {
    "i am": "you are",
    "i was": "you were",
    "i": "you",
    "i'm": "you are",
    "my": "your",
    "you are": "I am",
    "you were": "I was",
```

```
    "your": "my",
    "yours": "mine",
    "you": "me",
    "me": "you"
}

# Create a chatbot
pairs = [
    [
        r"my name is (.*)",
        ["Hello %1, how can I help you today?", "Hi there, %1. What can I do for you?"],
    ],
    [
        r"(.*) your name?",
        ["I'm just a chatbot. But you can call me ChatGPT.", "I don't have a name, I'm a chatbot."],
    ],
    [
        r"(.*) (hello|hi|hey)",
        ["Hello!", "Hi there!", "Hey! How can I assist you?"],
    ],
    [
        r"(.*) help (.*)",
        ["I'd be happy to help. What do you need assistance with?", "Sure, I'm here to help. What's your question?"],
    ],
    [
        r"(.*)",
        ["I'm sorry, I didn't quite understand that. Could you please rephrase?", "I'm not sure I follow. Can you clarify?"],
```

```
    ],
]

# Create and start the chatbot
chatbot = Chat(pairs, reflections)
chatbot.converse()

# Example of data augmentation
original_text = "I want to book a flight to New York"
augmented_text = augment_with_synonyms(original_text)
print("Original Text: ", original_text)
print("Augmented Text: ", augmented_text)
```