Code Documentation
Langchain + RAG + Ollama

by shivam dhumal

What is LangChain?

LangChain is a framework designed to help you create complex language model applications by chaining together various components. It allows you to build sophisticated workflows that leverage the power of large language models (LLMs) to perform tasks such as text generation, summarization, translation, and more.

History and Development

LangChain was developed to address the need for more structured and modular approaches to using LLMs. As AI applications grew in complexity, the need for a tool that could seamlessly integrate different language models and manage their interactions became evident. LangChain was born out of this necessity, evolving through community feedback and continuous improvements.

Core Functionalities

LangChain's core functionalities include:

- **Chaining Components**: You can connect different language models and tools to create end-to-end workflows.
- **Modularity**: LangChain supports modular development, allowing you to swap out components without affecting the entire system.

- **Integration**: It provides easy integration with various LLMs, enabling you to utilize the best model for each task.
- **Customization**: Offers extensive customization options to tailor workflows to your specific needs.

Example Use Cases

LangChain excels in scenarios where complex workflows involving multiple language models are required. Here are a few practical applications:

- **Content Generation**: Automate content creation by chaining together models for topic research, drafting, and editing.
- **Customer Support**: Build intelligent chatbots that can handle multi-step queries by integrating different models for understanding and response generation.
- **Data Analysis**: Combine models for data extraction, summarization, and reporting to streamline your analytics processes.

What is RAG?

RAG, or Retrieval-Augmented Generation, is a technique that combines the power of text generation models with information retrieval systems. By incorporating retrieval mechanisms, RAG enhances the generated text's

accuracy and relevance by pulling in contextual information from external sources.

History and Development

RAG was developed by Facebook AI Research to address the limitations of traditional text generation models, which often struggle with factual accuracy and coherence. By integrating retrieval systems, RAG aims to provide more grounded and contextually appropriate responses. The approach has gained traction for its ability to improve the quality of generated content significantly.

Core Functionalities

RAG's core functionalities include:

- **Information Retrieval**: Integrates with external knowledge bases to retrieve relevant information.
- **Enhanced Text Generation**: Uses retrieved data to produce more accurate and contextually relevant outputs.
- **Flexibility**: Can be applied to various tasks such as question answering, summarization, and content creation.
- **Scalability**: Efficiently handles large-scale data retrieval and generation tasks.

Example Use Cases

RAG is particularly useful in scenarios where the accuracy and relevance of generated content are crucial. Here are some practical applications:

- **Question Answering**: Improve the precision of answers by retrieving relevant documents from a knowledge base.
- **Content Creation**: Generate detailed and accurate articles by incorporating up-to-date information from trusted sources.
- **Customer Support**: Provide more accurate and contextually relevant responses by retrieving relevant information from a customer database or knowledge base.

Detailed Comparison

1. **Core Architecture**

LangChain Architecture

LangChain's architecture is designed to seamlessly integrate multiple language models and components into a unified workflow. Think of it as a modular pipeline where each component performs a specific task, and the output of one component feeds into the next. Here's a breakdown of the key components:

- **Input Processor**: This component handles raw input data, performing preprocessing tasks such as tokenization and normalization.
- **Model Chain**: The core of LangChain, where multiple language models are chained together. Each model in the chain performs a distinct function, such as text generation, sentiment analysis, or translation.
- **Output Processor**: After the models have processed the data, this component formats the output, ensuring it meets the desired specifications.

The workflow typically involves:

1. **Data Input**: Raw text or structured data is fed into the Input Processor.
2. **Processing**: The data flows through the Model Chain, with each model adding its layer of processing.
3. **Output Generation**: The final processed data is formatted and outputted by the Output Processor.

Example: Imagine you're building a content generation tool. You might start with a topic analysis model, pass the result to a text generator, and then to a sentiment analyzer to ensure the tone matches your requirements.

RAG Architecture

RAG combines the capabilities of retrieval systems with generative models. Its architecture consists of two main components:

- **Retriever**: This component searches for relevant documents or information from a predefined database or external sources. It uses advanced search algorithms to find the most pertinent data.
- **Generator**: After retrieving the information, this component uses a language model to generate coherent and contextually accurate text based on the retrieved data.

The workflow typically involves:

1. **Query Input**: A user query is inputted into the system.
2. **Information Retrieval**: The Retriever searches the knowledge base for relevant documents.
3. **Text Generation**: The Generator uses the retrieved information to produce a detailed and accurate response.

Example: In a customer support application, a user's question about a product can trigger the Retriever to find relevant documentation. The

Generator then creates a concise and informative answer using that documentation.

2. Key Features

LangChain Features

LangChain boasts several unique features:

- **Chaining of Models**: You can easily create complex workflows by chaining multiple language models.
- **Modularity**: Its modular design allows you to swap out models and components without disrupting the entire workflow.
- **Extensive Integration**: Supports integration with various LLMs and external APIs.
- **Customization**: Provides robust customization options to tailor each component to your specific needs.

RAG Features

RAG also offers distinct features:

- **Enhanced Accuracy**: By combining retrieval with generation, RAG improves the factual accuracy of generated text.

- **Contextual Relevance**: Retrieval mechanisms ensure the generated content is contextually appropriate and up-to-date.
- **Flexibility**: Can be applied to various tasks, from question answering to content creation.
- **Scalability**: Efficiently handles large-scale data and can retrieve information from extensive databases.

3. Use Cases

LangChain Use Cases

- **Automated Content Creation**: Chain models for topic research, drafting, and editing to automate the entire content creation process.
- **Intelligent Chatbots**: Develop chatbots that handle complex, multi-step interactions by chaining models for intent recognition, response generation, and sentiment analysis.
- **Data Processing Pipelines**: Create pipelines that perform sequential data processing tasks, such as extraction, transformation, and loading (ETL).

RAG Use Cases

- **Customer Support**: Enhance support systems with precise, contextually relevant responses by retrieving and generating accurate answers from a knowledge base.
- **Research Assistance**: Aid researchers by generating summaries and detailed explanations based on the latest research papers and documents.
- **Content Generation**: Produce well-informed articles and reports by retrieving relevant data and generating coherent content.

4. Performance and Scalability

LangChain

LangChain is designed to be highly efficient, but performance can vary based on the complexity of the model chain and the volume of data processed. It scales well with the addition of more powerful hardware or distributed computing resources.

Example: If you're processing large volumes of text for a news aggregation site, you'll appreciate LangChain's ability to handle multiple tasks in parallel.

RAG

RAG is built for high performance, particularly in tasks requiring factual accuracy and contextual relevance. Its dual-component architecture allows it to efficiently handle large datasets and complex queries.

Example: For a real-time Q&A system in a medical application, RAG's ability to retrieve and accurately generate responses ensures reliable and quick information delivery.

5. Ease of Implementation

LangChain

- **Installation**: Straightforward with well-documented installation procedures.
- **Configuration**: Highly configurable to suit various workflows and requirements.
- **Setup Complexity**: Moderate; involves chaining multiple models, which may require some initial setup and fine-tuning.

RAG

- **Installation**: Also straightforward with comprehensive documentation.

- **Configuration**: Flexible, with various options for integrating external knowledge bases.
- **Setup Complexity**: Moderate to high; setting up the retriever and ensuring it accesses the correct databases can be complex.

Real-World Applications

Case Studies

LangChain

- **Tech Company**: A tech startup used LangChain to automate content creation for their blog, significantly reducing the time spent on drafting and editing articles.
- **E-commerce Platform**: An e-commerce company implemented LangChain to build a customer service chatbot that handles product queries and support requests, improving response times and customer satisfaction.

RAG

- **Healthcare Provider**: A healthcare organization used RAG to develop a medical Q&A system that provides accurate, contextually relevant answers to patient queries based on the latest medical research.

- **Financial Services**: A financial firm implemented RAG to generate detailed reports and analyses, leveraging real-time data retrieval to ensure accuracy and relevance.

Success Stories

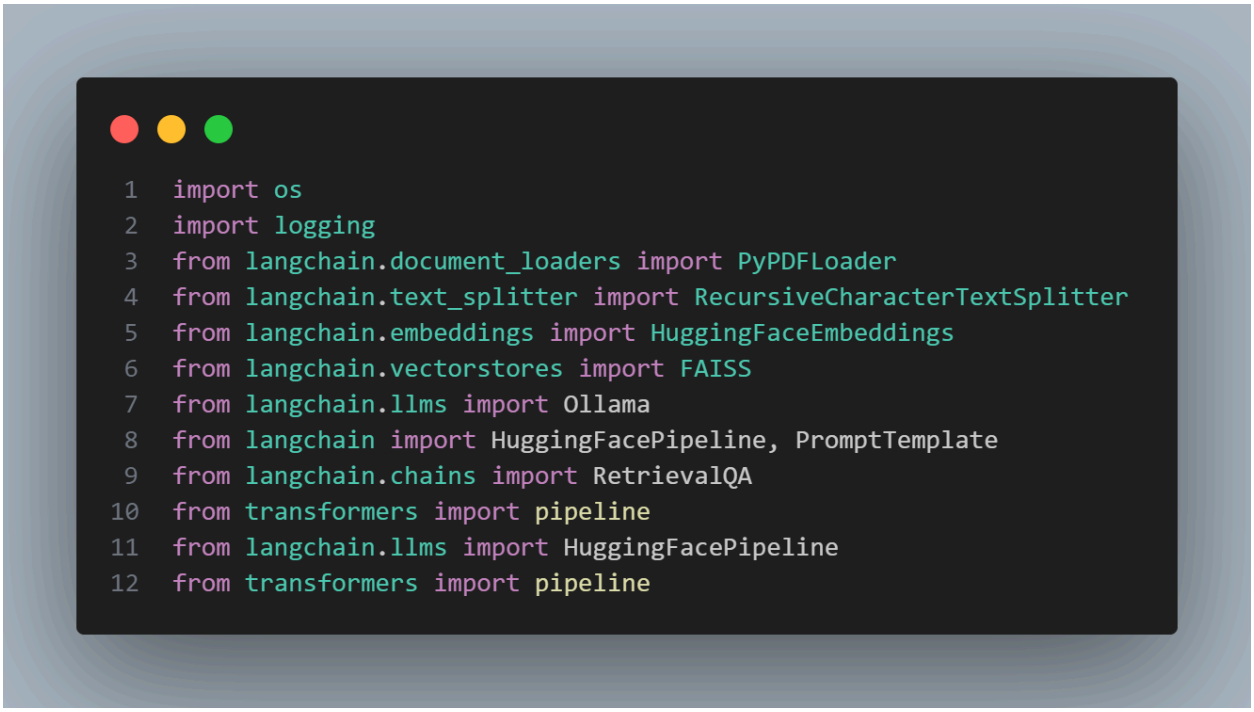
LangChain

- **Increased Efficiency**: Companies using LangChain reported a 40% increase in content production efficiency and a 30% reduction in manual editing efforts.
- **Enhanced Customer Engagement**: E-commerce platforms saw a 20% improvement in customer engagement and satisfaction through intelligent chatbots.

RAG

- **Improved Accuracy**: Healthcare providers noted a significant improvement in the accuracy of information provided to patients, enhancing trust and reliability.
- **Faster Decision-Making**: Financial analysts were able to make quicker, more informed decisions with reports generated by RAG, leading to better business outcomes.

Code



```
import os
import logging
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.llms import Ollama
from langchain import HuggingFacePipeline, PromptTemplate
from langchain.chains import RetrievalQA
from transformers import pipeline
from langchain.llms import HuggingFacePipeline
from transformers import pipeline
```

1. Importing Libraries

import os
import logging

- os: This is a built-in Python library that allows you to interact with the operating system, like handling file paths and environment variables.
- logging: This module is used for tracking events, errors, or general information in your code. It helps in debugging by recording logs, such as when the code runs, or if any issues arise.

2. Langchain Document Loading

from langchain.document_loaders import PyPDFLoader

- PyPDFLoader: A document loader class from Langchain that is used to load PDF files. It converts the content of PDF documents into text that can be processed further (e.g., for question answering).

3. Text Splitting

from langchain.text_splitter import RecursiveCharacterTextSplitter

- RecursiveCharacterTextSplitter: This is a text splitting method. When processing large texts, it's often useful to split the text into smaller chunks (called "documents" in Langchain). This class helps break down large documents into manageable pieces based on the number of characters.

4. Embeddings

from langchain.embeddings import HuggingFaceEmbeddings

- HuggingFaceEmbeddings: This is used to convert the text into numerical representations called embeddings. Embeddings capture the semantic meaning of the text in a format that machine learning models can understand and use for tasks like similarity search.

5. Vector Stores

from langchain.vectorstores import FAISS

- FAISS: This is a library for performing similarity search. It helps to store and search large collections of embeddings efficiently. For example, after converting a PDF into embeddings, you can use FAISS to quickly find similar text in a large collection of documents.

6. LLMs (Large Language Models)

from langchain.llms import Ollama

- Ollama: This is a class from Langchain used to interact with the Ollama language model, such as the Llama model. Ollama is a framework for hosting and interacting with language models, which are used to generate human-like text based on input prompts.

7. HuggingFace Pipeline

from langchain import HuggingFacePipeline, PromptTemplate

- HuggingFacePipeline: This integrates Hugging Face's model pipeline (like text generation or translation) with Langchain, so you can use pre-trained models from Hugging Face with Langchain’s higher-level abstractions.
- PromptTemplate: This class is used to define templates for text prompts, which are the input questions or commands that you give to language models. You can define how you want the prompt to be structured before passing it to the model.

8. Retrieval QA (Question Answering)

from langchain.chains import RetrievalQA

- RetrievalQA: This class creates a retrieval-based question-answering chain. It allows you to ask questions based on documents you have loaded. It retrieves relevant documents and passes them to the language model to generate an answer.

9. Transformers

from transformers import pipeline

- pipeline: This is a high-level API from the Transformers library by Hugging Face. It simplifies using pre-trained models for tasks like text generation, summarization, question-answering, etc.

10. Combining Everything: The Full Workflow

Here's how you could put everything together based on the imports and the tasks at hand:

- Load documents: You can load documents (like PDFs) using PyPDFLoader.
- Split large documents: You might want to split these documents into smaller chunks using RecursiveCharacterTextSplitter if they are too large.
- Convert text to embeddings: Use HuggingFaceEmbeddings to turn the text into numerical vectors that represent the meaning of the text.
- Store embeddings in a vector database: Use FAISS to store these embeddings in a way that allows for efficient similarity search.
- Use a Language Model (LLM): With Ollama, you can interact with a language model to generate answers, summaries, or completions based on the provided input.
- QA Chain: RetrievalQA allows you to ask questions based on the content loaded into the system. It retrieves the most relevant documents and generates answers using the LLM.

11. How the Components Interact

1. Document Loading and Preprocessing:
    - You first load your documents (such as PDFs).
    - Then, you split the content into smaller chunks, as LLMs work better with smaller contexts.
2. Creating Embeddings:
    - Convert the chunks of text into embeddings (numerical representations) using HuggingFaceEmbeddings.
3. Storing Embeddings in FAISS:
    - These embeddings are stored in FAISS, which allows you to quickly find the most relevant documents later when you need to answer questions.
4. Using an LLM for Question Answering:
    - The RetrievalQA chain is used to create a system where you can input a question, and it will find the most relevant chunks of text from your documents, process them with a language model (via Ollama), and generate an answer.

Example Scenario:

- You load a PDF document using PyPDFLoader.
- You split the text into smaller chunks using RecursiveCharacterTextSplitter.
- You generate embeddings for each chunk using HuggingFaceEmbeddings and store them in FAISS.
- You ask a question based on the documents.
- RetrievalQA retrieves the relevant chunks and uses Ollama to generate an answer.

```python
mport os
import logging
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.llms import Ollama
from langchain import HuggingFacePipeline, PromptTemplate
from langchain.chains import RetrievalQA
from transformers import pipeline
from langchain.llms import HuggingFacePipeline
from transformers import pipeline

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Folder containing PDFs
pdf_folder = 'C:\\Users\\dhuma\\Desktop\\books\\project Chatrapati'
vector_db_file = "vector_db.faiss"

# Step 1: Parse PDFs
all_documents = []
for filename in os.listdir(pdf_folder):
    if filename.endswith(".pdf"):
        pdf_path = os.path.join(pdf_folder, filename)
        logging.info(f"Processing PDF: {filename}")
        try:
            loader = PyPDFLoader(pdf_path)
            documents = loader.load()
            all_documents.extend(documents)
        except Exception as e:
            logging.error(f"Error reading PDF {filename}: {e}")

if not all_documents:
    raise ValueError("No documents found in the folder.")

# Step 2: Split Texts
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
chunks = text_splitter.split_documents(all_documents)
logging.info(f"Text split into {len(chunks)} chunks.")

# Step 3: Create Embeddings
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
if os.path.exists(vector_db_file):
    logging.info("Loading existing vector database.")
    vector_db = FAISS.load_local(vector_db_file, embedding_model, allow_dangerous_deserialization=True)
else:
    logging.info("Creating new vector database.")
    vector_db = FAISS.from_documents(chunks, embedding_model)
    vector_db.save_local(vector_db_file)

from langchain.llms import Ollama
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

# Step 4: Setup Ollama LLM using Langchain
llama_model_path = "llama3.2"  # The identifier for the Llama 3.2 model in Ollama

# Initialize Ollama LLM
llm = Ollama(model=llama_model_path)

# Example of creating a prompt template and chain with Ollama
prompt = PromptTemplate(input_variables=["question"], template="Answer this question: {question}")

```

This code sets up a complete workflow for processing documents, creating embeddings, and using a language model for question answering. Let's break it down and explain each part in detail:

1. Configuring Logging

```
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
```

- logging.basicConfig(): This function sets up the logging system for the application.
    - level=logging.INFO: This means the logging system will capture all logs of level INFO and higher (INFO, WARNING, ERROR, CRITICAL). INFO logs are used for general information, like steps in your process.
    - format='%(asctime)s - %(levelname)s - %(message)s': This specifies the format of the log messages. It includes the timestamp (asctime), the log level (levelname), and the log message (message).

Logging is crucial for debugging, tracking progress, and understanding what the program is doing at any point.

2. Folder Containing PDFs

```
pdf_folder = 'C:\\Users\\dhuma\\Desktop\\books\\project Chatrapati'
vector_db_file = "vector_db.faiss"
```

- pdf_folder: This variable holds the path to the folder that contains your PDF files. These PDFs are the documents you want to process.
- vector_db_file: This variable holds the name of the file where you will store the vector database (which stores the document embeddings). It uses the FAISS library for efficient vector-based search.

3. Parsing PDFs

```
all_documents = []
for filename in os.listdir(pdf_folder):
    if filename.endswith(".pdf"):
        pdf_path = os.path.join(pdf_folder, filename)
        logging.info(f"Processing PDF: {filename}")
        try:
            loader = PyPDFLoader(pdf_path)
            documents = loader.load()
            all_documents.extend(documents)
        except Exception as e:
            logging.error(f"Error reading PDF {filename}: {e}")
```

- os.listdir(pdf_folder): This gets the list of files in the pdf_folder.
- filename.endswith(".pdf"): This checks if the file is a PDF by looking at its extension.
- PyPDFLoader(pdf_path): This is part of Langchain, which loads the content from the PDF file. It converts the PDF into text documents that can be processed further.
- loader.load(): This loads the content of the PDF into a list of documents (which are chunks of text).
- all_documents.extend(documents): This adds the extracted documents to the all_documents list.
- logging.info(): This logs a message when a PDF is being processed.
- logging.error(): If there's an error while reading a PDF, this logs an error message.

This section processes all PDF files in the specified folder and converts them into text.

4. Splitting the Texts

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
chunks = text_splitter.split_documents(all_documents)
logging.info(f"Text split into {len(chunks)} chunks.")
```

- RecursiveCharacterTextSplitter: This class splits long texts (documents) into smaller chunks. Large text can be difficult for models to process, so splitting it into manageable sizes is important.
    - chunk_size=1000: This sets the maximum size of each chunk (1000 characters).
    - chunk_overlap=200: This ensures that each chunk has an overlap with the next chunk. It's useful for maintaining context across chunks.
- text_splitter.split_documents(all_documents): This splits all the loaded documents into smaller chunks.
- logging.info(f"Text split into {len(chunks)} chunks."): This logs the number of chunks the text has been split into.

This part ensures that large documents are split into smaller, more manageable chunks to feed into the model.

5. Creating Embeddings

```
embedding_model =
HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
if os.path.exists(vector_db_file):
    logging.info("Loading existing vector database.")
    vector_db = FAISS.load_local(vector_db_file, embedding_model,
allow_dangerous_deserialization=True)
else:
    logging.info("Creating new vector database.")
    vector_db = FAISS.from_documents(chunks, embedding_model)
```

vector_db.save_local(vector_db_file)

- HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2"): This initializes an embedding model using Hugging Face's pre-trained model "all-MiniLM-L6-v2". This model converts the text chunks into numerical vectors (embeddings) that represent the meaning of the text.
- FAISS.load_local(vector_db_file, embedding_model): This loads an existing vector database if it exists, which contains previously created embeddings.
- FAISS.from_documents(chunks, embedding_model): If no vector database exists, this creates a new one by converting the text chunks into embeddings using the specified embedding model.
- vector_db.save_local(vector_db_file): This saves the created vector database locally in a file named vector_db.faiss.

This part of the code creates numerical representations (embeddings) of the documents and stores them in a database for fast search and retrieval.

6. Setup Ollama LLM using Langchain

```
from langchain.llms import Ollama
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

llama_model_path = "llama3.2"  # The identifier for the Llama 3.2 model in Ollama

# Initialize Ollama LLM
llm = Ollama(model=llama_model_path)
```

- from langchain.llms import Ollama: This imports the Ollama class from Langchain, which is used to interact with the Llama language model (in this case, the version "llama3.2").
- llama_model_path = "llama3.2": This specifies the path or identifier for the Llama model.
- llm = Ollama(model=llama_model_path): This initializes the Llama model so you can interact with it to generate responses.

7. Creating a Prompt Template

```
prompt = PromptTemplate(input_variables=["question"], template="Answer this question: {question}")
```

- PromptTemplate: This class allows you to define a template for prompts, which are used to interact with language models. In this case, the template expects a question as input and generates a prompt like “Answer this question: {question}”.

```
prompt = PromptTemplate(input_variables=["question"], template="Answer this question: {question}")

# Create the chain with the prompt and LLM
qa_chain = LLMChain(llm=llm, prompt=prompt)

# Step 6: Ask Questions
query = "Who was shivaji mahraj?"  # Replace with your actual query
response = qa_chain.run(query)

# Print the answer
print(f"Answer: {response}")

```

This part of the code is focused on asking a question to the language model (Llama 3.2 via Ollama) and getting an answer based on the prompt and the processed documents. Let's break it down step by step to understand it clearly.

1. Define the Prompt Template

prompt = PromptTemplate(input_variables=["question"], template="Answer this question: {question}")

- PromptTemplate: This class is used to define how the input to the language model should be structured. You can think of it as a "blueprint" for the question you want to ask.
    - input_variables=["question"]: This tells the template that it expects an input variable named question. When you run the code, you will pass a question into this variable.
    - template="Answer this question: {question}": This is the actual template string that will be sent to the language model. {question} is a placeholder that will be replaced with the actual query you want to ask. For example, if you want to ask "Who was Shivaji Maharaj?", the template will transform into: "Answer this question: Who was Shivaji Maharaj?"

The PromptTemplate is used to format the query in a consistent way that the language model can understand.

2. Create the LLM Chain

qa_chain = LLMChain(llm=llm, prompt=prompt)

- LLMChain: This is a chain that ties together a language model (LLM) and a prompt template. It combines them to allow the language model to generate a response based on the input query.
    - llm=llm: This specifies the language model to use (in this case, the Llama model initialized earlier).
    - prompt=prompt: This specifies the prompt template that will be used to format the input query before sending it to the model.

So, the qa_chain object is essentially a "chain" that takes a query, formats it using the prompt, and then sends it to the Llama model for processing.

3. Ask a Question

```
query = "Who was Shivaji Maharaj?"  # Replace with your actual query
response = qa_chain.run(query)
```

- query = "Who was Shivaji Maharaj?": This defines the query (or question) that you want to ask the language model. You can replace this with any other question you want to ask.
- qa_chain.run(query): This runs the chain by taking the query, formatting it with the PromptTemplate, and sending it to the Llama model. The model will process the query and return a response.

The run() method essentially:

1. Takes the query (e.g., "Who was Shivaji Maharaj?").
2. Formats it into the prompt template (e.g., "Answer this question: Who was Shivaji Maharaj?").
3. Sends the formatted prompt to the language model.
4. Retrieves the model's answer.

4. Print the Answer

```
print(f"Answer: {response}")
```

- print(f"Answer: {response}"): This prints the response returned by the language model. The response variable contains the answer that the model generates based on the provided query.