LangChain - A Framework for LLM Applications

What is LangChain?

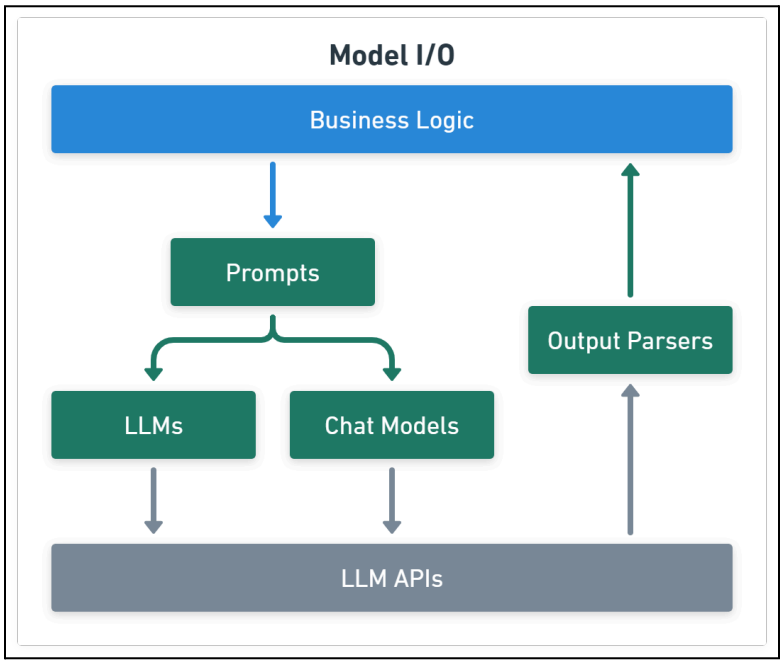
LangChain is an open-source framework designed to simplify the development of applications using Large Language Models (LLMs). It provides tools to **connect LLMs with external data sources**, **manage memory**, **chain multiple calls**, and **integrate with various APIs**.

Why Use LangChain?

- Simplifies LLM integration into applications
- Provides components for structured conversation flows
- Enables memory management for contextual interactions
- Supports external data sources like databases and APIs
- Offers tools for retrieval-augmented generation (RAG)

Official Documentation: View

Simple and Easy Visualization to understand LangChain:



Steps to Set Up LangChain with OpenAI:

Note: You can choose any model but for understanding examples we'll use OpenAI.

Step 1: Install Required Packages:

```
pip install python-dotenv
pip install langchain-openai langchain-community
```

Step 2: Set Up OpenAI API Key:

```
import os
from dotenv import load_dotenv
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")
```

Step 3: Generate Text Using LangChain + OpenAI:

```
from langchain_openai import ChatOpenAI

# Initialize openai chat-model
llm = ChatOpenAI() # here you can also pass name of opanai model like ChatOpenAI(model="gpt-4o-mini")

# invoke your query
response = llm.invoke('suggest any movie title')

print(response.content) # extract content which is your exact answer
```

```
"Echoes of the Past"
```

Models and Temperature for ChatModels:

Models: The AI model used for generating responses,
example: `ChatOpenAI(model="gpt-4o-mini")`

Note: If no model is specified, LangChain defaults to using `gpt-3.5-turbo`.

Temperature: Controls randomness in responses; lower values (e.g., 0.2) make output deterministic, while higher values (e.g., 0.8) add creativity.
example: `ChatOpenAI(model="gpt-4o-mini", temperature=0.7)`

Messages:

In LangChain, **messages** represent a list of interactions between the user and the AI, categorized by **roles** and message **content** types.

1. **Roles:**
   We have many defined roles for a particular model below are some examples

   - **system**: Sets behavior or instructions for the model.
     Example: `{"role": "system", "content": "You are a math tutor."}`

   - **user**: Represents the user's input.
     Example: `{"role": "user", "content": "What is 2+2?"}`

   - **assistant**: Stores the model's responses.
     Example: `{"role": "assistant", "content": "The answer is 4."}`

2. **Content:**
   In LangChain, the main types of **messages** are:

   - **SystemMessage:** for content which should be passed to direct the conversation.
     Example: `SystemMessage(content="You are a coding assistant.")`

   - **HumanMessage:** for content in the input from the user.
     Example: `HumanMessage(content="How do I reverse a list in Python?.")`

   - **AIMessage:** for content in the response from the model.
     Example: `AIMessage(content="you can use list[::-1] to reverse a list.")`

Prompt Templates:

A **PromptTemplate** in LangChain formats prompts dynamically by replacing placeholders with input values.

We have different types of promptTemplates but mostly we use String PromptTemplate and ChatPromptTemplate.
.

String PromptTemplate:
These prompt templates are used to format a single string, and generally are used for simpler inputs

Code example for String PromptTemplate:

```
from langchain_core.prompts import PromptTemplate

# Define the prompt template
template = "Suggest a movie title based on the genre: {genre}"

# Format the prompt with a specific genre
prompt = PromptTemplate.from_template(template)
formatted_prompt = prompt.invoke(input="action") # we get movies title based on "action" genre

# Invoke the model with the formatted prompt
response = llm.invoke(formatted_prompt)

# Print the response
print(response.content)
```

```
"Strike Force"
```

Note: In the above code , the prompt is dynamically formatted using the genre as input, and the model generates a movie title suggestion based on it.

Chat PromptTemplate:

These prompt templates are used to format a list of messages. These "templates" consist of a list of templates themselves

Code example for Chat PromptTemplate:

```
from langchain_core.prompts import ChatPromptTemplate

# Define the prompt template
template = [
    ("system", "You are a movie recommendation assistant."),
    ("user", "Suggest only a movie title based on the genre: {genre}")
]


# Format the prompt with a specific genre
prompt = ChatPromptTemplate(template)
formatted_prompt = prompt.invoke(input="action") # we get movies title based on "action" genre

# Invoke the model with the formatted prompt
response = llm.invoke(formatted_prompt)

# Print the response
print(response.content)
```

```
"Mad Max: Fury Road"
```

Note: In the above code ,The `ChatPromptTemplate` now includes two parts: a **system** message to set the assistant's role and a **user** message asking for a movie suggestion based on the genre.

Chains, Runnables, and Runnable Types in LangChain:

Chains:

A **Chain** is a sequence of operations (e.g., prompt creation, model invocation, and output parsing) combined to perform a task.

Runnable:

A **Runnable** is an executable unit in LangChain, making it easy to build and compose workflows.

Code example:

```
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Define the prompt template
template = "Suggest only a movie title based on the genre: {genre}"

# Create the PromptTemplate
prompt = PromptTemplate.from_template(template)

# Define the output parser
output_parser = StrOutputParser()

# Create the chain
movie_title_chain = prompt | llm | output_parser

# Invoke the chain with correct input
response = movie_title_chain.invoke({"genre": "action"})

# Print the response
print(response)
```

```
"Excessive Force"
```

Note: In the above code , `movie_title_chain = prompt | llm | output_parser` creates a **chain** by linking a prompt template, a language model, and an output parser, ensuring smooth data flow from input to output.

Composed Chains:

It combines multiple chains together, allowing complex workflows by linking smaller chains sequentially or in parallel.

Code example for composed chains:

```
# Define the prompt template
template = "Suggest 2-3 lines short movie summary for movie: {movie_title}"

# Create the PromptTemplate
movie_summary_prompt = PromptTemplate.from_template(template)

# Define the output parser
output_parser = StrOutputParser()

# Create the composed chain
composed_chain = {"movie_title":movie_title_chain} | movie_summary_prompt | llm | output_parser

# Invoke the chain with correct input
response = composed_chain.invoke({"genre": "action"})

# Print the response
print(response)

"""
Output

As a group of astronauts aboard a spaceship hurtles towards the edge of the universe,
they must confront their inner demons and trust each other to survive.
But when mysterious incidents start occurring on board, they realize there may be a traitor among them.
Will they be able to uncover the truth before it's too late? Set against the backdrop of space exploration,
"The Edge of Destruction" is a gripping tale of suspense, betrayal, and the fight for survival.

"""
```

Note: In the above code , we have combined 2 different chains (`movie_title_chain` and `movie_summary_prompt`) into one composed chain so it first generate a movie title based on the genre, then use that title to get a short movie summary,

Runnable Types:

LangChain provides different types of runnables to structure workflows:

RunnableLambda → Wraps a simple function as a runnable.

RunnableMap → Runs multiple functions in parallel.

RunnablePassthrough → Passes data through without modification.

RunnableParallel → Runs multiple tasks concurrently.

The widely and most used one is **RunnableLambda** which is used to write **our own custom function** as runnable.

Code example for RunnableLambda:

```
from langchain_core.runnables import RunnableLambda

# Create the composed chain with a lambda function in the middle
composed_chain = (
    movie_title_chain
    | RunnableLambda(lambda x: print("Movie Title is : ", x) or x)  # Print the movie title and return it
    | RunnableLambda(lambda x: {"movie_title": x})  # Lambda function to process movie title
    | movie_summary_prompt
    | llm
    | output_parser
)

# Invoke the chain with correct input
response = composed_chain.invoke({"genre": "action"})

# Print the response
print(response)

"""
Output
Movie Title is :  "Explosive Pursuit"
When a notorious criminal steals a dangerous weapon of mass destruction,
a skilled detective is on a race against time to track him down before
it's too late. As the chase intensifies, both men must confront their
own personal demons and face off in a thrilling showdown that will
determine the fate of thousands. Explosive action and heart-pounding
suspense await in this high-stakes pursuit.

"""
```

Note: In the above code, we have used `RunnableLambda` to process and transform data within the composed chain. First, it prints the generated movie title for debugging purposes, then converts it into the required format `{ "movie_title": x }` before passing it to the next stage. This ensures the output of `movie_title_chain` seamlessly integrates with `movie_summary_prompt`, generating a short movie summary based on the title.

Types of Chaining in Langchain:

We usually have 3 types of widely used chaining.

1. Sequential Chain or Extended Chaining –The output of one step is passed as the input to the next in a linear order. (Example: Movie title → Movie summary generation.)

> **Note:** All the examples we have seen so far demonstrate **Sequential Chaining**, where the output of one step is passed as the input to the next step in a linear order. This ensures a structured flow where each layer processes data one by one before passing it to the next stage.

2. Parallel Chaining – Multiple chains run independently at the same time, and their outputs can be combined later. (Example: Generating both a movie summary and cast details simultaneously.)

2. Conditional Chaining –The next step depends on certain conditions or logic based on the previous output. (Example: If the genre is "sci-fi," fetch space-related movies; otherwise, fetch general recommendations.)

RunnableSequence, RunnableParallel and RunnableBranch:

RunnableSequence: It is used to define a structured pipeline where multiple runnables (LLMs, prompts, output parsers, etc.) are executed in a specific order. It ensures smooth data flow from one step to the next.

Code example for RunnableSequence:

```python
from langchain_core.runnables import RunnableSequence

# Create the composed chain with a RunnableSequence
composed_chain = RunnableSequence(
    prompt , llm , output_parser
)

# Invoke the chain with correct input
response = composed_chain.invoke({"genre": "action"})

# Print the response
print(response)
```

```
"Force of Fury"
```

Note: In the above code , we have used **RunnableSequence** and separated each execution with a **comma** (,) instead of **pipe operator** (|).

RunnableParallel: It allows multiple runnables to execute simultaneously and returns their outputs together, enabling parallel processing for efficiency.

Code example for RunnableParallel:

```python
from langchain_core.runnables import RunnableParallel

# Define the prompt template
hindi_movie_template = "give 1 line conclusion of movie: {movie_title} in hindi"
spanish_movie_template = "give 1 line conclusion of movie: {movie_title} in spanish"

# Create the PromptTemplate
hindi_movie_summary_prompt = PromptTemplate.from_template(hindi_movie_template)
spanish_movie_summary_prompt = PromptTemplate.from_template(spanish_movie_template)

# Define the output parser
output_parser = StrOutputParser()

# Create chains
hindi_movie_chain = {"movie_title":movie_title_chain} | hindi_movie_summary_prompt | llm | output_parser
spanish_movie_chain = {"movie_title":movie_title_chain} | spanish_movie_summary_prompt | llm | output_parser

# Create the composed chain
composed_chain = RunnableParallel(hindi_conclusion = hindi_movie_chain, spanish_conclusion = spanish_movie_chain)

# Invoke the chain with correct input
response = composed_chain.invoke({"genre": "action"})

# Print the response
print("Hindi conclusion: ",response["hindi_conclusion"]) # hindi conclusion
print() # for new line
print("Spanish conclusion: ",response["spanish_conclusion"]) # spanish conclusion
```

```
Hindi conclusion:  "रैपिड फायर" एक एक्शन-पैक्ड थ्रिलर फिल्म है जो आपको नॉन-स्टॉप एक्शन का अनुभव कराती है।

Spanish conclusion:  La batalla final demostró que el verdadero poder reside en la unión y el trabajo en equipo.
```

RunnableBranch: It dynamically selects and executes one runnable based on input conditions, allowing conditional logic in the chain execution.

Code example for RunnableParallel:

```python
from langchain_core.runnables import RunnableBranch

# Define the prompt templates
hindi_movie_template = "Give a 1-line conclusion of the movie: {movie_title} in Hindi."
spanish_movie_template = "Give a 1-line conclusion of the movie: {movie_title} in Spanish."
default_template = "Give a 1-line conclusion of the movie: {movie_title} in English."

# Create the PromptTemplate objects
hindi_movie_summary_prompt = PromptTemplate.from_template(hindi_movie_template)
spanish_movie_summary_prompt = PromptTemplate.from_template(spanish_movie_template)
default_summary_prompt = PromptTemplate.from_template(default_template)

# Define the output parser
output_parser = StrOutputParser()

# Create chains
hindi_movie_chain = hindi_movie_summary_prompt | llm | output_parser
spanish_movie_chain = spanish_movie_summary_prompt | llm | output_parser
default_movie_chain = default_summary_prompt | llm | output_parser

# Define the branching logic based on language preference
composed_chain = RunnableBranch(
    (lambda x: x.get("language") == "hindi", hindi_movie_chain),
    (lambda x: x.get("language") == "spanish", spanish_movie_chain),
    default_movie_chain  # Default case if no condition matches
)
```

```python
# Invoke the chain with different inputs
response_hindi = composed_chain.invoke({"movie_title": "Inception", "language": "hindi"})
response_spanish = composed_chain.invoke({"movie_title": "Inception", "language": "spanish"})
response_default = composed_chain.invoke({"movie_title": "Inception", "language": "french"})  # Defaults to English

# Print responses
print("Hindi Conclusion:", response_hindi)
print("Spanish Conclusion:", response_spanish)
print("Default Conclusion:", response_default)
```

```
Hindi Conclusion: इंसान की मनोरंजकता और परिपक्षियों की प्रासंगिक समीक्षा - इन्सेप्शनमें खोज।
Spanish Conclusion: La película Inception es una emocionante aventura surrealista que desafía la percepción de la realidad.
Default Conclusion: Inception blurs the line between reality and dreams, leaving viewers questioning their own perception of reality.
```