# LangChain

LangChain is designed for developers creating applications powered by large language models (LLMs). It provides a comprehensive set of pre-built components, making it easier to perform common tasks in this domain.

LLM (Large Language Model) Abstraction is a crucial element in LangChain. It serves as an abstraction layer over different LLM implementations. This abstraction allows the same code to be compatible with models from various providers such as OpenAI, Hugging Face, Google, and Anthropic.

Model Types

* Completion Models - These models take a string input and provide text that semantically extends the input. They are useful for tasks where the goal is to expand or complete a given text input.
* Chat-Oriented Models - Designed for conversations, these models can take a dialogue between a user and AI and provide semantic completions or answers. They are particularly suited for interactive applications.

LangChain - a standalone component that allows for the composition of multiple components into a coherent whole. These are referred to as "chains."

Chain - a sequence of calls to various components, which can include other chains. It's a method to stitch together multiple functions and operations to achieve a desired output.

## Demo LLM Chain

**from** langchain.prompts **import** PromptTemplate

model\_name = "gpt-3.5-turbo"

temperature = 0.8

llm = OpenAI(model\_name=model\_name, temperature=temperature, max\_tokens = 500)

prompt\_template = PromptTemplate.from\_template(

"""Act as Marvin, a robot from Douglas Adams' Hithiker Guide.

Tell me a {story\_type} about the person described in context below.

Context: {context}"""

)

text = prompt\_template.format(story\_type="funny joke", context="I'm a software engineer learning to use large language models")

**print**(text)

**print**("====OUTPUT=====\n")

output = llm(text)

**print**(output)

## Document Loaders

A langchain Document Loader is an abstraction used for loading langchain documents from various data sources (like databases, CSV files, Wikipedia, etc.).

'page\_content' field - This is a class for unstructured content such as plain text. It holds the actual data, and a metadata array, which is a collection of data that provides information about other data.

CSVLoader() - a specific tool designed to read and interpret CSV files and construct a CSV loader object.

CSVLoader().loader is responsible for the actual reading and processing of the data from the CSV file. The output is shown to be a langchain document object, which includes both the review content and metadata about the source file and the specific row from which the review was taken.

## Output parsers

Large language models (LLMs) can be used to generate output which can then be parsed into structured objects. Key terms highlighted include:

Unstructured Text - the raw output of LLMs, which is typically not in a format that's easy for developers to work with.

Structured Output - the preferred format for developers, such as JSON or CSV. This format is easier to use and integrate into applications.

Parsing - the process of converting the unstructured text output from LLMs into structured objects.

The integration of Parsers allows developers to convert LLM output into formats that can be more easily manipulated and integrated into different applications.

## Retrieval Augmented Generation (RAG)

Retrieval Augmented Generation (RAG), is a technique that enhances the capabilities of large language models (LLMs). RAG can integrate a company's data, like a knowledge base, with LLMs. This allows applications to leverage both the power of LLMs and the specific information contained in the company's own data.

1. The process begins with a user query, which is used to search a vector database. Vector databases are used to store data. These databases are essential for adding additional, semantically relevant information to the LLM
2. The system then retrieves documents that are semantically closest to the query.
3. The retrieved documents are passed along with the original query to the LLM. This provides the LLM with extra context and up-to-date information, resulting in a more informed and accurate response.
4. Document transformers are used to prepare data by breaking it into smaller chunks. This is beneficial for indexing large documents and achieving a more precise match between the user's query and the document content.
5. Text Embedding Models convert document chunks into embeddings that capture the semantic meaning of the data.
6. Vector Storage is where the embeddings are stored, ready for retrieval.
7. Retrievers fetch the semantically relevant chunks for the LLM to process.

The RAG system allows for efficient use of LLMs by not overwhelming them with data and ensures precise and accurate output.

**from** langchain.indexes **import** VectorstoreIndexCreator

**from** langchain.document\_loaders.csv\_loader **import** CSVLoader

loader = CSVLoader(file\_path='./tv-reviews.csv')

index = VectorstoreIndexCreator().from\_loaders([loader])

query = "Based on the reviews in the context, tell me what people liked about the picture quality"

index.query(query)

## LangChain Memory Component

The LangChain memory component is a solution designed to address the limitations of LLMs in maintaining conversational continuity.

LLMs are stateless and they typically process each interaction in isolation, lacking memory of past exchanges. This presents challenges:

* A limited context window.
* The absence of conversation history in sequential interactions.

Memory helps overcome context limitations by storing and retrieving context as needed, enabling LLMs to remember important details beyond their immediate context window.

## Simulating Maintained State

The LangChain Memory Component addresses both the issue of maintaining ongoing conversation state and summarizing dialogue.

* Allows for recalling the entire conversation within a manageable size or summarizing it as it grows.
* Retains relevance and context in long interactions.
* Ensures LLMs can access summarized versions to maintain continuity without losing context.
* Enhances user input with context before executing core logic and to store interactions for future reference after the response.

## ConversationBufferMemory

The ConversationBufferMemory component is essential for maintaining a history of chat messages, enabling the chatbot to have a contextual understanding of the ongoing conversation.

* When a user sends a message, it is added to the conversation history, and similarly, the chatbot's responses are also recorded. This helps in providing context for the chatbot's subsequent messages.
* The conversation buffer memory can output the chat history either as a continuous stream or as a list.
* The stored string of chat messages provides essential context for the language model (LM) when generating its next message.

## Types of Chains in AI

1. Router Chain

* Purpose - Allows dynamic, non-deterministic sequences of operations based on previous step outputs.
* Application - Adapts interactions with a large language model based on context or question type.
* Example - A chatbot handling customer inquiries about electronics, software, and home appliances. It identifies the inquiry type and routes to the appropriate response mechanism.

1. Sequential Chain

* Purpose - Facilitates a series of operations where the output of one is the input for the next.
* Application - Useful for processes needing multiple language model calls or functions in a specific order.
* Example - Creating a product description followed by generating a review using that description.

1. Transformation Chain

* Purpose - Involves altering or processing inputs at various points in an operational sequence.
* Application - Used to create pipelines for specific tasks.
* Example - Extracting video transcripts, stripping timestamps, and then summarizing the text.
* Each of these chains serves a distinct purpose, enabling more sophisticated and context-sensitive applications of AI models in various scenarios.