DAY2 :

Main Use Cases

- **Summarization** - Express the most important facts about a body of text or chat interaction
- **Question and Answering Over Documents** - Use information held within documents to answer questions or query
- **Extraction** - Pull structured data from a body of text or an user query
- **Evaluation** - Understand the quality of output from your application
- **Querying Tabular Data** - Pull data from databases or other tabular source
- **Code Understanding** - Reason about and digest code
- **Interacting with APIs** - Query APIs and interact with the outside world
- **Chatbots** - A framework to have a back and forth interaction with a user combined with memory in a chat interface
- **Agents** - Use LLMs to make decisions about what to do next. Enable these decisions with tools.

A Chain with Memory is a sequence of actions and observations that can be used to perform a specific task. In contrast, an Agent with Memory is a conversational agent that can engage in a conversation with the user and utilize memory to provide context-aware responses.

Here are the main differences between an Agent with Memory and a Chain with Memory:

- **Functionality:** A Chain with Memory is designed to perform a specific task or sequence of actions, while an Agent with Memory is designed to engage in a conversation and provide context-aware responses.
- **Conversation Flow:** An Agent with Memory uses the **AgentExecutor** class to handle the conversation flow, allowing it to interact with the user and respond to their queries. On the other hand, a Chain with Memory follows a predefined sequence of actions and observations without user interaction.
- **Memory Usage:** An Agent with Memory utilizes memory to remember the history of the conversation and use that information to provide more accurate responses. It can remember previous questions and answers, providing context-aware responses based on the conversation context. A Chain with Memory may also use memory, but its usage is limited to the specific task or sequence of actions it is designed for.

An Agent with Memory is a conversational agent that can converse with the user and utilize memory to provide context-aware responses. It differs from a Chain with Memory, a sequence of actions and observations designed for a specific task.

To add memory to an Agent, the following steps can be followed:

- Create an `LLMChain` with memory.
- Use the `ConversationBufferMemory` class to store and retrieve conversation history.
- Construct the `LLMChain` with the Memory object.
- Create the Agent using the `ZeroShotAgent` class, passing in the `LLMChain` and the memory object.
- Use the `AgentExecutor` class to execute the Agent and handle the conversation flow.

Module III : Agents

Agents

The language model that drives decision making.

More specifically, an agent takes in an input and returns a response corresponding to an action to take along with an action input

Agents in LangChain

Agents in LangChain are systems that use a language model to interact with other tools.

They can be used for tasks such as grounded question/answering, interacting with APIs, or taking action. LangChain provides:

Agents vs. Chains

The core idea of agents is to use an LLM to choose a sequence of actions.

In chains, a sequence of actions is hardcoded (in code). In agents, a language model is used as a reasoning engine to determine which actions to take and in which order. An agent uses a language model to interact with other tools or environments.

Agents involve a language model:

- Making decisions about which actions to take.
- Taking those actions.
- Observing the results.
- Repeating the process until a desired outcome is achieved.

An agent is different from a chain in that a chain is a sequence of calls, whether to a language model or another utility.

A chain focuses on the **flow of information and computation**. In contrast, an agent focuses on **decision-making and interaction with the environment**.

Agents can be used for applications such as personal assistants, question answering, chatbots, querying tabular data, interacting with APIs, extraction, summarization, and evaluation.

Agents use an LLM as a reasoning engine and connect it to two key components: tools and memory.

Agents involve a language model:

- Making decisions about which actions to take.
- Taking those actions.
- Observing the results.
- Repeating the process until a desired outcome is achieved.

The Agent is the core component responsible for decision-making. It harnesses the power of a language model and a prompt to determine the next steps to achieve a specific objective. The inputs to an agent typically include:

- **Tools:** Descriptions of available tools (more on this later).
- **User Input:** The high-level objective or query from the user.
- **Intermediate Steps:** A history of (action, tool output) pairs executed to reach the current user input.

What are tools and toolkits?

In LangChain, tools and toolkits provide additional functionality and capabilities to agents.

Tools are individual components that perform specific tasks, such as retrieving information from external sources or processing data.

Conversely**, toolkits** are collections of tools designed to work together and provide a more comprehensive set of functionalities.

Tools

A 'capability' of an agent. This is an abstraction on top of a function that makes it easy for LLMs (and agents) to interact with it. Ex: Google search.

Why do agents even need tools?

Providing an agent with the right tools becomes a powerful system that can execute and implement solutions on your behalf.

Combining an agent’s decision-making abilities with the functionality provided by tools allows it to perform a wide range of tasks effectively.

Here are a few reasons why an agent needs tools:

- **Access to external resources:** Tools allow an agent to access and retrieve information from external sources, such as databases, APIs, or web scraping. This enables the agent to gather relevant data and use it for decision-making.
- **Data processing and manipulation:** Tools provide the necessary functionality for an agent to process and manipulate data. This includes cleaning and transforming data, performing calculations, or applying machine learning algorithms.
- **Integration with other systems:** Tools enable agents to integrate with other systems or services. For example, an agent may need to interact with a chatbot platform, a customer relationship management (CRM) system, or a knowledge base. Tools facilitate this integration and allow agents to exchange information with these systems.
- **Customization and extensibility:** While LangChain provides built-in tools, it also allows users to define custom tools. This means an agent can be equipped with tools tailored to its unique requirements. Custom tools can be created to address specific tasks or to integrate with proprietary system

AgentExecutor

The `AgentExecutor` class is responsible for executing the agent's actions and managing the agent's memory.

It takes an agent, a set of tools, and an optional memory object as input.

The `AgentExecutor` provides a more flexible and customizable way to run the agent, as you can specify the tools and memory to be used.

When to use `AgentExecutor`:

- When you want more control over executing the agent's actions and memory management.
- When you want to specify the tools and memory to be used by the agent.

Initialize Agent

The `initialize_agent` function is a convenience function provided by LangChain that simplifies creating an agent.

It takes the agent class, the language model, and an optional list of tools as input.

It automatically initializes the agent with the specified language model and tools.

When to use `initialize_agent`:

- When you want a simplified way to create an agent without specifying the memory.
- When you want to create an agent with default settings, quickly.

If you need more customization and control over the agent's execution, you should use `AgentExecutor`.

If you prefer a more straightforward and quicker way to create an agent, you can use `initialize_agent`.

Why does an agent need memory?

An agent in LangChain needs memory to store and retrieve information during decision-making.

Memory allows an agent to maintain context and remember previous interactions, which is crucial for providing personalized and coherent responses.

Here are a few reasons why an agent needs memory:

- **Contextual understanding:** Memory helps an agent understand the context of a conversation or interaction. By storing previous messages or user inputs, the agent can refer back to them and provide more accurate and relevant responses. This allows the agent to maintain a coherent conversation and understand the user's intent.
- **Long-term knowledge:** Memory enables an agent to accumulate knowledge over time. By storing information in memory, the agent can build a knowledge base and use it to answer questions or provide recommendations. This allows the agent to provide more informed and accurate responses based on past interactions.
- **Personalization:** Memory allows an agent to personalize its responses based on the user's preferences or history. By remembering previous interactions, the agent can tailor its responses to the specific needs or interests of the user. This enhances the user experience and makes the agent more effective in achieving its objectives.
- **Continuity:** Memory ensures continuity in a conversation or interaction. The agent can pick up where it left off by storing the conversation history and maintaining a consistent dialogue with the user. This creates a more natural and engaging user experience.

Before we dive into building the agent, it's essential to revisit some key terminology and schema:

- **AgentAction:** This is a data class representing the action an agent should take. It consists of a `tool` property (the name of the tool to invoke) and

a `tool_input` property (the input for that tool).

- **AgentFinish:** This data class indicates that the agent has finished its task and should return a response to the user. It typically includes a dictionary of return values, often with a key "output" containing the response text.
- **Intermediate Steps:** These are the records of previous agent actions and corresponding outputs. They are crucial for passing context to future iterations of the agent

Chains :

What are chains?

Chains refer to sequences of calls — whether to an LLM, a tool, or a data preprocessing step.

a **chain** functions similar to a necklace, where each bead contributes a unique element to the overall structure. Chains have various components that the Lang Chain library offers, making our work with LLMs easier and more efficient.

Why we need chains?

Chains combine LLMs with other components, creating applications by executing a sequence of functions.

Components of chains in LangChain

- **LLMs (Large Language Models):** These are core processing units used for tasks like generation, translation, summarization, and question answering.
- **Memory:** Stores information between nodes or across chain runs for context continuity.
- **Tools:** Perform specific tasks like summarization, sentiment analysis, or data retrieval.
- **Agents:** Act as decision-makers, choosing which tools to use based on input and context.

- **Data Retrieval Components:** Fetch data from internal databases, external APIs, or local files.

What are Chains in LangChain?

A chain is an end-to-end wrapper around multiple individual components executed in a defined order.

Chains are one of the core concepts of LangChain. Chains allow you to go beyond just a single API call to a language model and instead chain together multiple calls in a logical sequence.

They allow you to combine multiple components to create a coherent application.

Some reasons you may want to use chains:

- To **break down a complex task** into smaller steps that can be handled sequentially by different models or utilities. This allows you to leverage the different strengths of different systems.
- **To add state and memory between calls.** The output of one call can be fed as input to the next call to provide context and state.
- To add **additional processing, filtering or validation logic between calls.**
- For easier **debugging** and instrumentation of a sequence of calls.

Foundational chain types in LangChain

The `LLMChain`, `RouterChain`, `SimpleSequentialChain`, and `TransformChain` are considered the core foundational building blocks that many other more

complex chains build on top of. They provide basic patterns like chaining LLMs, conditional logic, sequential workflows, and data transformations.

- `LLMChain`: Chains together multiple calls to language models. Useful for breaking down complex prompts.
- `RouterChain`: Allows conditionally routing between different chains based on logic. Enables branching logic.
- `SimpleSequentialChain`: Chains together multiple chains in sequence. Useful for linear workflows.
- `TransformChain`: Applies a data transformation between chains. Helpful for data munging and preprocessing.
- **summarization chain :**
- **sequential chain :**

LLMChain

The most commonly used type of chain is an LLMChain.

The LLMChain consists of a PromptTemplate, a language model, and an optional output parser. For example, you can create a chain that takes user input, formats it with a PromptTemplate, and then passes the formatted response to an LLM. You can build more complex chains by combining multiple chains, or by combining chains with other components.

The main differences between using an LLMChain versus directly passing a prompt to an LLM are:

- LLMChain allows **chaining multiple prompts together**, while directly passing a prompt only allows one. With LLMChain, you can break down a complex prompt into **multiple more** straightforward prompts and chain them together.
- LLMChain maintains s**tate and memory between prompts**. The output of one prompt can be fed as input to the following prompt to provide context. Directly passing prompts lack this memory.

- LLMChain makes adding preprocessing logic, validation, and instrumentation between prompts easier. This helps with debugging and quality control.

Creating an LLMChain

To create an LLMChain, you need to specify:

- The language model to use
- The prompt template

  Use `apply` when you have a list of inputs and want to get the LLM to generate text for each one, it will run the LLMChain for every input dictionary in the list and return a list of outputs.

  `generate` is similar to apply, except it returns an LLMResult instead of a string.

  Use `predict` when you want to pass inputs as keyword arguments instead of a dictionary. This can be convenient if you don't want to construct an input dictionary.

  Use `LLMChain.run` when you want to pass the input as a dictionary and get the raw text output from the LLM.

Router Chains

Router chains allow **routing inputs to different destination chains based on the input text**. This allows the building of chatbots and assistants that can handle diverse requests.

- Router chains examine the input text and route it to the appropriate destination chain
- Destination chains handle the actual execution based on the input
- Router chains are powerful for **building multi-purpose chatbots/assistants**

Sequential Chains

There are two types of sequential chains:

1) `SimpleSequentialChain`: The simplest form of sequential chains, where each step has a singular input/output, and the output of one step is the input to the next.

2) `SequentialChain`: A more general form of sequential chains allows multiple inputs/outputs.

SimpleSequentialChain

The simplest form of a sequential chain is where each step has a single input and output.

The output of one step is passed as input to the next step in the chain. You would use `SimpleSequentialChain` it when you have a linear pipeline where each step has a single input and output. `SimpleSequentialChain` implicitly passes the output of one step as input to the next.

This is great for composing a precise sequence of LLMChains where each builds directly on the previous output.

When to use:

- You have a clear pipeline of steps, each with a single input and output
- Each step builds directly off the previous step's output

- Useful for simple linear pipelines with one input and output per step.
- Create each step as an `LLMChain`.
- Pass list of `LLMChains` to `SimpleSequentialChain`.
- Call `run()` passing the initial input.

How to use:

1) Define each step as an `LLMChain` with a single input and output

2) Create a `SimpleSequentialChain` passing a list of the LLMChain steps

3) Call `run()` on the SimpleSequentialChain with the initial input

SequentialChain

A more general form of sequential chain allows **multiple inputs and outputs per step.**

You would use `SequentialChain` when you have a more complex pipeline where steps might have multiple inputs and outputs.

`SequentialChain` allows you to explicitly specify all the input and output variables at each step and map outputs from one step to inputs of the next. This provides more flexibility when steps might have multiple dependencies or produce multiple results to pass along.

When to use:

- You have **a sequence of steps but with more complex input/output requirements**
- You need to track m**ultiple variables across steps in the chain**

How to use

- Define each step as an LLMChain, specifying multiple input/output variables
- Create a SequentialChain specifying all input/output variables
- Map outputs from one step to inputs of the next
- Call run() passing a dict of all input variables
- The key difference is `SimpleSequentialChain` handles implicit variable passing whereas SequentialChain allows explicit variable specification and mapping.

When you would use SequentialChain vs SimpleSequentialChain

Use `SimpleSequentialChain` for **linear sequences with a single input/output.**
Use `SequentialChain` for more **complex sequences** with multiple inputs/outputs.

The key difference

`SimpleSequentialChain` is for linear pipelines with a **single input/output per s**tep. Implicitly passes variables.

`SequentialChain` handles more complex pipelines with **multiple inputs/outputs** per step. Allows explicitly mapping variables.

This uses a standard ChatOpenAI model and prompt template. You chain them together with the `|` operator and then call it with `chain.invoke`. We can also get async, batch, and streaming support out of the box.

Transformation

Transformation Chains allows you to define custom data transformation logic as a step in your LangChain pipeline. This is useful when you must preprocess or transform data before passing it to the next step.

Memory in LangChain:

In LangChain, the Memory module is responsible for persisting the state between calls of a chain or agent, which helps the language model remember previous interactions and use that information to make better decisions.

It provides a standard interface for persisting state between calls of a chain or agent, enabling the language model to have memory and context.

What does it do?

The Memory module enables the language model to have memory and context, allowing the LLM to make informed decisions.

It allows the model to remember user inputs, system responses, and any other relevant information. The stored information can be accessed and utilized during subsequent interactions.

Why do I need it?

The Memory module helps you build more interactive and personalized applications.

It gives the language model a sense of continuity and memory of past interactions. With memory, the model can provide more contextually relevant responses and make informed decisions based on previous inputs.

When do I use it?

You should use the Memory module whenever you want to create applications that require **context and persistence between interactions.**

It is handy for tasks like **personal assistants,** where the model needs to remember user **preferences, previous queries, and other relevant information.**

Every memory system performs two main tasks: reading and writing.

Every chain has core logic that requires specific inputs.

Some inputs originate from the user, while others derive from memory. During a run, a chain accesses its memory system twice:

1) It reads from memory to supplement user inputs before executing core logic.

2) After processing but before responding, it writes the current run's data to memory for future reference.

Incorporating Memory

Two fundamental decisions shape any memory system:

1) The method of storing state.

2) The approach to querying that state.

Storing: At the heart of memory lies a record of all chat interactions.

LangChain's memory module offers various ways to store these chats, ranging from temporary in-memory lists to enduring databases.

Querying: While storing chat logs is straightforward, designing algorithms and structures to interpret them isn't.

A basic memory system might display recent messages. A more advanced one could summarize the last 'K' messages. The most refined systems might identify entities from stored chats and present details only about those entities in the current session.

Different applications demand unique memory querying methods.

LangChain's memory module simplifies the initiation with basic systems and supports creating tailored systems when necessary.

Implementing Memory

1) Setup prompt and memory

2) Initialize LLMChain

3) Call LLMChain

ConversationBufferMemory

. `ConversationBufferMemory` is a simple memory type that stores chat messages in a buffer and passes them to the prompt template.

You can inspect variables by calling `memory.load_memory_variables`

`load_memory_variables` returns a single key, `history`.

This means that your chain (and likely your prompt) expects an input named `history`. You control this variable through parameters on the memory class. For example, if you want the memory variables to be returned in the key `chat_history` you can do the following:

Memory as strings, or list of strings

When it comes to memory, one of the most common types is the storage and retrieval of chat messages.

There are two ways to retrieve these messages:

1) A single string that concatenates all the messages together, which is useful when the messages will be passed in Language Models

2) A list of ChatMessages, which is useful when the messages are passed into ChatModels.

ConversationBufferWindowMemory

The `ConversationBufferWindowMemory` is a tool that keeps track of past interactions in a conversation.

It does this by maintaining a list of the most recent interactions, and only using the last K interactions. This helps to ensure that the buffer doesn't become too large and allows for a sliding window of the most recent interactions to be kept. This type of memory is beneficial for keeping the history of past interactions small and manageable.

```
memory.load_memory_variables({}).  # use to check the memory varables
```