# Langchain

→ framework for building applications based on LLM
→ can create things like RAG, chatbot etc.
→ lets get started with Langchain with hands on example

## (1) Simple Ollama Connection

→ Install ollama

```
pip install langchain
```

```
pip install langchain-community
```

→ Write a basic code (very basic)

```
from langchain_community.llms import Ollama

llm = Ollama(model="deepseek-r1:14b")

response = llm.invoke("Tell me a joke")
print(response)
```

→ Run the code

```
<think>

</think>

Sure! Here's a light-hearted joke for you:

Why don't skeletons fight each other?
Because they don't have the *guts*! 😄
```

## (2) Using a different library

→ When running the above code, I got

a warning to using a different library for interfacing with ollama using langchain.

→ following shows the new library, its installation and output

- Installation

```
% pip install langchain_ollama
```

- Code updates

```
from langchain_ollama import OllamaLLM

llm = OllamaLLM(model="deepseek-r1:14b")

response = llm.invoke("Tell me a joke")
print(response)
```

- Output is the same, however.

```
<think>

</think>

Sure! Here's a light-hearted joke for you:

Why don't skeletons fight each other?
Because they don't have the *guts*! 😄
```

(3) Creating a chat software

→ class to use for chat is "ChatOllama".

→ we are going to use chatollama to create

an LLM model and feed it with message to anticipate an output from the LLM.

→ The entire code is as follows:

```python
# import library for Ollama
from langchain_ollama import ChatOllama
from langchain_core.messages import HumanMessage, SystemMessage

# create variables to connect to Ollama server
OLLAMA_MODEL_NAME = "deepseek-r1:14b"
OLLAMA_BASE_URL = "http://localhost:11434"

# create llm model
llm = ChatOllama(
    model=OLLAMA_MODEL_NAME,
    base_url=OLLAMA_BASE_URL
)

# let us create a message parameter to send to LLM
messages = [
    SystemMessage(content="You are a helpful and concise AI assistant."),
    HumanMessage(content="Write a short story in less than 100 words.")
]

try:
    response = llm.invoke(messages)
    print(f"Your query: {messages[1].content}")
    print(f"Response from LLM: {response.content}")
except Exception as e:
    print(f"An error occurred: {str(e)}")

print(f"Program terminating")
```

import

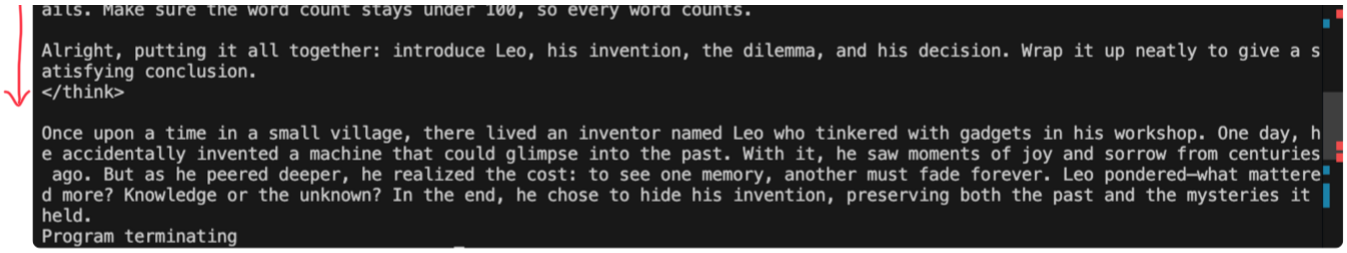
variable to define LLM

create LLM model

send prompt to LLM

→ Following is the output that is returned from the LLM

```
Your query: Write a short story in less than 100 words.
Response from LLM: <think>
Okay, so the user asked me to write a short story in less than 100 words. Hmm, they probably want something quick and engagi
ng. Maybe they're looking for a creative snippet or just a fun read.

I should consider the theme. Something universal like time travel could resonate with many people. It's interesting and has
a lot of possibilities within a tight word limit.

The main character... a curious inventor named Leo sounds relatable. He can be portrayed as friendly and driven by his passi
on, which makes him likable.

What's the conflict?发明时光机后发现可以回到过去，但必须做出牺牲。This adds depth to the story. It shows that even with grea
t inventions, there are consequences or trade-offs.

The ending should leave a lasting impression. Instead of using the machine for personal gain, Leo chooses to give it up. Thi
s highlights selflessness and the importance of keeping the machine hidden, leaving room for imagination.

I need to keep the language concise but vivid. Each sentence should contribute to building the story without unnecessary det
```




→ In the output above, the think portion is significantly longer than the actual output

④ creating a chat AI with history

→ Above example uses a simple LLM object model and invokes it using an array of messages

→ now let us look at a bit complex example where we will use history and conversation_chain to create a chat loop.

→ some important concepts before we start

- Chatprompt Template
  - it is a structured prompt definition
  - allows definition of prompt by providing

list of message template which can either be static message or of SystemMessage, HumanMessage or AIMessage type

- allows invoke or format calls to create final prompt/messages to be constructed with provided input variables

- ConversationChain
  - langchain design for conversational AI
  - takes in LLM, memory and prompt

- ConversationBufferMemory:
  - a memory type in Langchain
  - primary function to store raw, unsummarized history of conversation as it happens
  - keeps a complete record of all inputs from human and outputs in the conversation
  - when input needs to be provided to LLM, it retrieves the stored message and formats them to inject them to LLM's prompt

→ following a code of an AI assistant chat

server ~ which only quits when "quit" is explicitly typed

→ TODO