LangChain: Multi-Turn Chat

In this notebook, we build a short conversation by **manually tracking history**. Think of history as a list you keep growing. Each turn, we send the whole list back to the model.

Why this matters:

- It preserves context across turns
- Follow-up questions make sense without repeating yourself
- It is the core idea behind chat memory in larger apps

This pattern is the foundation for chat memory, multi-step workflows, and more advanced LangChain apps.

Loading environmental variables

Before we do anything, let's load our .env file so the API key is available.

```
# Import the dotenv loader
from dotenv import load_dotenv
# Import the chat model initializer
from langchain.chat_models import init_chat_model
# Import explicit message types for a conversation
from langchain.messages import SystemMessage, HumanMessage, AIMessage

# Load variables from .env into the process environment
load_dotenv()
```

```
True
```

> **Output note:** If you see True, the .env file was found. False means it was not - easy fix is to check the filename and path.

Initialising the chat model and history

We begin by spinning up a lightweight model.

```
# Create a lightweight chat model for this demo
model = init_chat_model(
    # Choose a small, fast OpenAI model
    model="gpt-4o-mini"
)
```

Quick demo: the model does NOT remember by itself

We will call the model twice in succession. Notice how it cannot recall the name we gave it.

```
# First call: tell the model your name
first = model.invoke("My name is Roger.")
print("Call 1:", first.content)

# Second call: ask the model to recall the name
second = model.invoke("\nWhat is my name?")
print("Call 2:", second.content)
```

```
Call 1: Hi Roger! How can I assist you today?
Call 2: I don't have access to personal information about users unless it has been shared in
the course of our conversation. If you'd like to share your name or need help with something
else, feel free to let me know!
```

Output note: The second reply usually says it doesn't know or guesses. That is the point: without history, the model has **no memory**.

Setting the system instruction

So, let's give our model some memory. We start with an empty history list. This list will hold **System**, **Human**, and **AI** messages in order.

The first entry into our history list will be a `SystemMessage` that controls the assistant's behavior.

```
# Start an empty conversation history
history = []

# Add a system instruction to guide the assistant
history.append(
    SystemMessage(
        # Keep responses short and focused
        content="You are a concise tutor. Keep answers within 30 words."
    )
)

# Show the history after the first message
print(f"First memory:\n\n{history}")
```

```
First memory:

[SystemMessage(content='You are a concise tutor. Keep answers within 30 words.', additional_k
wargs={}, response_metadata={})]
```

Output note: You should see a list with a single SystemMessage. It looks a bit nerdy because it's a Python object, not a chat bubble.

Turn 1: Ask the first question

We add a `HumanMessage` , send the full history to the model, and then store the reply as an `AIMessage` .

```
# Add the user's first question
history.append(
    HumanMessage(
        # First user query
        content="What is overfitting?"
    )
)

# Send the full history to the model
ai_1 = model.invoke(history)

# Store the assistant response in history
history.append(
    AIMessage(
        # Capture just the model's text
        content=ai_1.content
    )
)

# Print the assistant's reply
print("Assistant:", ai_1.content)
```

```
Assistant: Overfitting occurs when a model learns the training data too well, capturing noise
and outliers, resulting in poor performance on unseen data.
```

Output note: You should get a short definition of overfitting. If it's long, your system rule might not have been applied.

Turn 2: Follow-up with context

We add another `HumanMessage` that depends on the first answer, and send the **entire history** again.

```
# Add a follow-up question that depends on context
history.append(
    HumanMessage(
        # Second user query
        content="Give one practical way to reduce it."
    )
)

# Send the updated history to the model
ai_2 = model.invoke(history)

# Store the assistant response in history
history.append(
    AIMessage(
        # Capture just the model's text
        content=ai_2.content
    )
)

# Print the assistant's reply
print("Assistant:", ai_2.content)
```

```
Assistant: Use cross-validation to evaluate model performance and ensure it generalizes effec
tively to new data.
```

Output note: Expect a concise, practical tip (often regularization). Still short - that's the system rule at work.

Reviewing the full conversation

Finally, we pretty-print every message in order.

```python
# Pretty-print every message in the conversation
for h in history:
    # Show each message in a readable format
    h.pretty_print()
```

```
================================ System Message ================================

You are a concise tutor. Keep answers within 30 words.
================================ Human Message =================================

What is overfitting?
================================== Ai Message ==================================

Overfitting occurs when a model learns the training data too well, capturing noise and outlie
rs, resulting in poor performance on unseen data.
================================ Human Message =================================

Give one practical way to reduce it.
================================== Ai Message ==================================

Use cross-validation to evaluate model performance and ensure it generalizes effectively to n
ew data.
```

Output note: You should see a clean, role-labeled transcript (System -> Human -> AI -> Human -> AI).

Key takeaways

- Multi-turn chat = keep a **history list** and resend it each turn.
- Store AI responses as `AIMessage` so the model can use them later.
- System rules apply to every turn as long as they stay in history.