

```
= Chat Models
:order: 4
:type: lesson
```

In the previous lesson, you learned how to communicate with the OpenAI LLM using Langchain.

The communication was simple, you provided a prompt and got a response.

In this lesson, you will learn how to use a chat model to `_have a conversation_` with the LLM.

== Chat Models vs Language Models

Until now, you have been using a language model to communicate with the LLM.

A language model predicts the next word in a sequence of words. Chat models are designed to have conversations - they accept a list of messages and return a conversational response.

Chat models typically support different types of messages:

- * System - System messages instruct the LLM on how to act on human messages
- * Human - Human messages are messages sent from the user
- * AI - Responses from the AI are called AI Responses

== Create a Chat Model

You are going to create an application that uses a chat model.

The application will:

- * Use a system message to provide instructions
- * Use a human message to ask a question
- * Receive an AI response to the question

Create a new Python program, import the required LangChain modules and instantiate the chat model.

[IMPORTANT]

.OpenAI API Key

Remember to update the API key with your own.

```
[source,python]
```

```
----
include::code/chat-model.py[tag=import]
```

```
include::code/chat-model.py[tag=llm]
```

```
----
```

Create a system message to provide instructions to the chat model using the

link:https://api.python.langchain.com/en/latest/messages/langchain_core.messages.system.SystemMessage.html[`SystemMessage`^] class.

```
[source,python]
```

```
----
```

```
include::code/chat-model.py[tag=system]
```

```
----
```

Create a

link:https://api.python.langchain.com/en/latest/messages/langchain_core.messages.human.HumanMessage.html[`HumanMessage`^] object to ask a question.

```
[source,python]
```

```
----
```

```
include::code/chat-model.py[tag=human]
```

```
----
```

You can now call the chat model passing a list of messages.

In this example, pass the system message with the instructions and the human message with the question.

```
[source,python]
```

```
----
```

```
include::code/chat-model.py[tag=invoke]
```

```
----
```

```
[%collapsible]
```

```
.Reveal the complete code
```

```
====
```

```
[source,python]
```

```
----
```

```
include::code/chat-model.py[tag=**]
```

```
----
```

```
====
```

The `response` is an

link:https://api.python.langchain.com/en/latest/messages/langchain_core.messages.ai.AIMessage.html[`AIMessage`^] object.

```
[source,python]
```

```
AIMessage(content="Dude, the weather is totally gnarly! It's sunny with some epic offshore winds. Perfect conditions for shredding some sick waves!", additional_kwargs={}, example=False)
```

```
=== Wrapping in a Chain
```

You can create a reusable chat model chain using what you have learned about prompts and chains.

Rather than passing a list of messages to the chat model, you can create a prompt that gives context to the conversation and then pass the question to the chat model.

Review this program and identify the following:

- . The prompt is created by combining `system` and `human` messages.
- . The chain is created using the chat model, the prompt and an output parser.
- . The question is passed to the chat model as a parameter of the `invoke` method.

```
[source,python]
----
include::code/chat-model-chain.py[tag=**]
----
```

Creating a chain is the first step to creating a more sophisticated chat model.

You can use chains to combine different elements into one call and support more complex features.

=== Giving context

Previously, you learned about grounding and how it can provide context to the LLM and avoid `_Hallucination_`.

Currently, the chat model is not grounded; it is unaware of surf conditions on the beach.

It responds based on the question and the LLMs training data (which could be months or years out of date).

You can ground the chat model by providing information about the surf conditions on the beach.

Review this example where the chat model can access current beach conditions (``current_weather``) as a system message (``context``) in the prompt.

```
[source,python]
----
include::code/chat-model-context.py[]
----
```

Run the program and predict what the response will be.

```
[%collapsible]
.Click to reveal the response
====
```

Below is a typical response. The LLM has used the context passed in the prompt to provide a more accurate response.

```
Dude, the surf at Watergate Bay is pumping! We got some sick 3ft waves rolling in, but unfortunately, we got some onshore winds messing with the lineup. But hey, it's all good, still plenty of stoke to be had out there!
====
```

Investigate what happens when you change the context by adding additional beach conditions.

Providing context is one aspect of Retrieval Augmented Generation (RAG). In this program, you manually gave the model context; however, you could have retrieved real-time information from an API or database.

== Check Your Understanding

```
include::questions/1-message-types.adoc[leveloffset=+1]
```

```
[.summary]
```

== Lesson Summary

In this lesson, you learned that chat models are designed for conversations and how to add additional context.

In the next lesson, you will learn how to give your chat model a memory so it can retain information between questions.