LangChain: Simple Chat Model

Welcome! In this notebook, we'll build the **smallest possible LangChain chat example** and then peek under the hood to see what the model *actually* returns.

We'll:

- load environment variables
- initialise a lightweight chat model
- send a quick prompt
- inspect the `AIMessage` object
- convert it into a plain Python dictionary
- and grab the text in the cleanest way possible

By the end, you'll know exactly what a LangChain chat response looks like — and how to work with it like regular Python data.

Loading environmental variables

Before we do anything, it's a good idea to load our environment variables.

The `python-dotenv` library reads this file and adds the variables to the process environment, making them accessible to our model via `os.getenv(...)` .

```python
# Import the dotenv loader
from dotenv import load_dotenv

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

```
True
```

Next, we want to create a **chat model** instance using LangChain.

Initialising the chat model

The `init_chat_model` helper function selects and configures the appropriate chat model based on the arguments provided. In this case, we are using a lightweight OpenAI model suitable for simple examples and demonstrations.

The returned `model` object acts as a callable interface to the underlying language model.

```python
# Import the helper to initialize a chat model
from langchain.chat_models import init_chat_model

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

Sending a prompt to the model

With the chat model initialised, we can now send it an input prompt (e.g., **"Hiya!"**).

The `invoke()` method executes a single model call and returns the model's response.

```python
# Send a simple prompt to the model
response = model.invoke(input="Hiya!")
```

Now, let's check out the model's response.

```python
# Display the raw AIMessage object
response
```

```
AIMessage(content='Hi there! How can I assist you today?', additional_kwargs={'refusal': Non
e}, response_metadata={'token_usage': {'completion_tokens': 10, 'prompt_tokens': 10, 'total_t
okens': 20, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0,
'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tok
ens': 0, 'cached_tokens': 0}}, 'model_provider': 'openai', 'model_name': 'gpt-4o-mini-2024-07
-18', 'system_fingerprint': 'fp_29330a9688', 'id': 'chatcmpl-D1HbVuqzrvbHqIp0TQTIx8GcbQnwZ',
'service_tier': 'default', 'finish_reason': 'stop', 'logprobs': None}, id='lc_run--019bec70-b
845-7483-bd84-00a843dcce74-0', tool_calls=[], invalid_tool_calls=[], usage_metadata={'input_t
okens': 10, 'output_tokens': 10, 'total_tokens': 20, 'input_token_details': {'audio': 0, 'cac
he_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}})
```

Inspecting the model response

When we call a chat model in LangChain, the result is **not just a plain string**.

Instead, LangChain returns an **`AIMessage` object**. This object contains:

- the assistant's text reply (`content`)
- metadata about the call (model name, token usage, finish reason, etc.)
- tool-call fields (empty in this simple example)

At first glance, this can look confusing because it's a rich Python object, not a "chat bubble".

To confirm what it actually is, let's check its Python type:

```python
# Check the Python type of the response
type(response)
```

```
langchain_core.messages.ai.AIMessage
```

Think of `langchain_core.messages.ai.AIMessage` as a fully-qualified “address” for a Python class inside the LangChain codebase.

It can be broken down like this:

- `langchain_core` → a Python **package**
- `messages` → a **sub-package** / module namespace inside langchain_core related to chat messages
- `ai` → a **module** (a .py file) inside messages that defines AI-related message types
- `AIMessage` → a **class** defined in that module

So, `AIMessage` is a class, located at:

package `langchain_core` → subpackage `messages` → module `ai` → class `AIMessage`

BUT...!

Didn't we only install the `langchain` package, and not `langchain_core` ?

Yes, but when we install `langchain` , we *implicitly* install `langchain-core` . We can prove this below:

```python
# Import the core package to inspect its version
import langchain_core
# Print the installed langchain_core version
print(langchain_core.__version__)
```

```
1.2.7
```

Converting the response into a plain Python dictionary

Now, let’s take the `AIMessage` object returned by the model and convert it into a **standard Python dictionary**.

LangChain message objects are built using **Pydantic**, which means they provide a `.model_dump()` method. This method extracts all fields from the object into a plain data structure that is easier to inspect, log, or serialise.

```python
# Import pretty-printer for readable output
from pprint import pprint
# Convert the AIMessage into a plain Python dict
response_dict = response.model_dump()
# Pretty-print the dictionary for inspection
pprint(response_dict)
```

```
{'additional_kwargs': {'refusal': None},
 'content': 'Hi there! How can I assist you today?',
 'id': 'lc_run--019bec70-b845-7483-bd84-00a843dcce74-0',
 'invalid_tool_calls': [],
 'name': None,
 'response_metadata': {'finish_reason': 'stop',
                       'id': 'chatcmpl-D1HbVuqzrvbHqIp0TQTIx8GcbQnwZ',
                       'logprobs': None,
                       'model_name': 'gpt-4o-mini-2024-07-18',
                       'model_provider': 'openai',
                       'service_tier': 'default',
                       'system_fingerprint': 'fp_29330a9688',
                       'token_usage': {'completion_tokens': 10,
                                       'completion_tokens_details': {'accepted_prediction_tok
ens': 0,
                                                                     'audio_tokens': 0,
                                                                     'reasoning_tokens': 0,
                                                                     'rejected_prediction_tok
ens': 0},
                                       'prompt_tokens': 10,
                                       'prompt_tokens_details': {'audio_tokens': 0,
                                                                 'cached_tokens': 0},
                                       'total_tokens': 20}},
 'tool_calls': [],
 'type': 'ai',
 'usage_metadata': {'input_token_details': {'audio': 0, 'cache_read': 0},
                    'input_tokens': 10,
                    'output_token_details': {'audio': 0, 'reasoning': 0},
                    'output_tokens': 10,
                    'total_tokens': 20}}
```

Understanding the dictionary output

This is the same model response as before, but now represented as a **plain Python dictionary** rather than a LangChain object.

Key fields to notice:

- **`content`**
  The actual text generated by the model.

- **`type`**
  Indicates the role of the message. Here it is `"ai"`, meaning the response came from the model.

- **`response_metadata`**
  Information about how the response was generated, including:
  - the model used
  - why the generation stopped (`finish_reason`)
  - token usage for the prompt and completion
- **`usage_metadata`**
  A simplified summary of input and output tokens, useful for tracking cost and performance.

- **`tool_calls / invalid_tool_calls`**
  Empty in this example because no tools were used.

At this stage, there is nothing “LangChain-specific” about the data structure — it is just ordinary Python data that can be logged, stored, or converted to JSON.

For example, we can extract the content using:

```python
# Pull out just the assistant's text content
response_dict["content"]
```

```
'Hi there! How can I assist you today?'
```

Pretty-printing the response

Because `response` is an `AIMessage` object (not a plain string), printing it directly can look noisy and hard to read.

LangChain provides a built-in helper method, `.pretty_print()`, which formats the message in a clean, human-readable way. This is especially useful in notebooks and live demos.

```python
# Pretty-print the response for readability
response.pretty_print()
```

```
================================== Ai Message ==================================

Hi there! How can I assist you today?
```

This output shows the assistant’s message clearly, without any of the surrounding metadata.

Behind the scenes, the `AIMessage` object still contains additional information such as token usage and model details — `.pretty_print()` simply presents the most relevant part for humans: the generated text.

Accessing the message content directly

If all you care about is the text generated by the model, you can access it directly using the `content` attribute.

This is often the most convenient option when building simple applications or when you want behaviour that feels like a traditional chatbot.

```python
# Print the response content as a plain string
print(response.content)
```

```
Hi there! How can I assist you today?
```

Here, the response is returned as a plain Python string.

At this point, all of the additional metadata has been ignored, and we are working only with the assistant’s text output. This is typically what you would pass into downstream application logic or display to an end user.