LangChain: Explicit Message Types

Welcome! In this notebook, we move beyond a single string prompt and use explicit message objects. This makes the roles clear and gives the model stronger guidance.

Why this matters:

- System rules stay separate from user input
- Behavior is easier to steer and debug
- The structure scales to history, tools, and chains

This message-based approach really shines when you build more sophisticated LangChain apps: **multi-turn chats** (with history), **chains**, and other higher-level workflows all build on the same idea.

Loading environmental variables

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

```python
# 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 (system + human)
from langchain.messages import SystemMessage, HumanMessage

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

```
True
```

> **Output note:** If you see True, the .env file was found. If it says False, double-check the filename and location.

Initialising the chat model

Next, we spin up a lightweight model for a quick demo.

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

Defining explicit message types

Instead of a single string, we build a tiny transcript made of message objects.

- **SystemMessage** sets the assistant's behavior, tone, and constraints.
- **HumanMessage** represents the user's actual request.

- **AIMessage** is the assistant's output (useful to store for later turns).

Order matters: system message first, then user message, then the AI reply.

This structure makes conversations easier to extend and debug.

```python
# Build an explicit list of messages
messages = [
    # SystemMessage sets behavior and tone
    SystemMessage(
        # Instruction the model should follow
        content="You are a helpful assistant who answers in two bullet points"
    ),
    # HumanMessage is the user's input
    HumanMessage(
        # The actual question we want answered
        content="What is the difference between a function and a method in Python?"
    )
]
```

Sending the messages

We pass the list into `model.invoke(...)` and get back an `AIMessage`. We then pretty-print it so it reads like a clean chat response.

```python
# Send the message list to the model
response = model.invoke(messages)

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

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

- A **function** is a block of code that performs a specific task and can be called independe
ntly. It is defined using the `def` keyword and can take parameters but is not associated wit
h any object.

- A **method** is similar to a function but is associated with an object and is defined withi
n a class. It typically operates on data contained within the object (its attributes) and is
called using the instance of the class.
```

Output note: You should see a neat two-bullet reply. Nice - that's your SystemMessage steering the format.

Key takeaways

- Explicit message types give you **control** and **clarity**.
- System = rules, Human = request, AI = response.
- The ordered list is the foundation for multi-turn chat.