Unit 2 - Part 1a: LangChain Setup & Models

1. Introduction: Why LangChain?

Before we write code, you might ask: **"Why not just use the Gemini API directly?"**

Imagine you build an app using OpenAI. Six months later, you want to switch to Gemini because it's cheaper.

- **Without LangChain:** You have to rewrite all your API calls (different endpoint, different parameters, different response format).
- **With LangChain:** You change **one line of code**.

LangChain is a **framework** that provides a standard interface for any Language Model. It's like a universal adapter for AI.

2. Concept: Tokens (The Atom of AI)

Models don't read words. They read **Tokens**. A token can be a word, part of a word, or even a space.

Example:

- Text: `"Hello World"`
- Tokens: `[101, 2055, 309]` (Hypothetical IDs)

Cost & Context

You pay per token. The model has a limit on how many tokens it can remember (Context Window).

3. Setting Up the Environment

We need two main libraries:

1. `langchain`: The core logic.
2. `langchain-google-genai`: The specific connector for Google models.

Let's install them quietly.

```
In [1]: %pip install python-dotenv --upgrade --quiet langchain langchain-google-genai
```

```
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/111.7 kB ? eta -:--:--
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 111.7/111.7 kB 3.4 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/66.5 kB ? eta -:--:--
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 66.5/66.5 kB 4.6 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/500.5 kB ? eta -:--:--
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╸ 491.5/500.5 kB 22.9 MB/s eta 0:00:0
1
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 500.5/500.5 kB 13.9 MB/s eta 0:00:0
0
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/158.1 kB ? eta -:--:--
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 158.1/158.1 kB 12.2 MB/s eta 0:00:0
0
```

4. Securely Loading API Keys

Never hardcode your API keys (e.g., `api_key = "AIzaSy..."`) in a notebook. If you share the notebook, your key is stolen.

Best Practice: Use `getpass` to enter it securely every session, or load it from environment variables.

```
In [4]: from dotenv import load_dotenv
        load_dotenv()

        import getpass
        import os

        if "GOOGLE_API_KEY" not in os.environ:
            os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter your Google API Key:
```

```
Enter your Google API Key: ··········
```

5. The Architecture (Flowchart)

What happens when we run the code below?

6. The `Temperature` Parameter (Critical Thinking)

When we initialize a model, we aren't just "turning it on". We are configuring its brain.

The most important setting is `temperature`.

- **Range:** 0.0 to 1.0 (sometimes higher).
- **Meaning:** How "random" should the choice of the next word be?

Let's create **two** versions of the same model to compare them side-by-side.

In [5]:
```
from langchain_google_genai import ChatGoogleGenerativeAI

# Model A: The "Accountant" (Precision)
llm_focused = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.0

# Model B: The "Poet" (Creativity)
llm_creative = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=1.
```

7. Experiment: Consistency vs. Creativity

We will ask both models to "Define the word 'Idea' in one sentence." We will run the code **TWICE** for each model.

Hypothesis:

- The Focused model (Temp=0) should say the *exact same thing* both times.
- The Creative model (Temp=1) should say *different things*.

In [6]:
```
prompt = "Define the word 'Idea' in one sentence."

print("--- FOCUSED (Temp=0) ---")
print(f"Run 1: {llm_focused.invoke(prompt).content}")
print(f"Run 2: {llm_focused.invoke(prompt).content}")
```

```
--- FOCUSED (Temp=0) ---
Run 1: An idea is a thought, concept, or mental image formed in the mind.
Run 2: An idea is a thought, concept, or suggestion that is formed or exists in
the mind.
```

In [7]:
```
print("--- CREATIVE (Temp=1) ---")
print(f"Run 1: {llm_creative.invoke(prompt).content}")
print(f"Run 2: {llm_creative.invoke(prompt).content}")
```

```
--- CREATIVE (Temp=1) ---
Run 1: An idea is a thought, concept, or mental impression formed in the mind.
Run 2: An idea is a thought or suggestion about a possible course of action, or
a concept that arises in the mind.
```

8. Conclusion for Part 1a

What did we learn?

1. **LangChain** abstracts the messy API details.
2. **Tokens** are the currency of AI.
3. **Temperature** is a control knob for randomness.

In the next notebook (**1b**), we will look at how to control the *Input* using Prompt Templates.

Unit 2 - Part 1b: Prompts & Parsers

1. Introduction: The Pipeline

Real AI apps are not just `print(llm.invoke("hi"))`. They are pipelines.

The Data Flow (Flowchart)

In [8]:
```
# Setup from Part 1a (Hidden for brevity)
from dotenv import load_dotenv
load_dotenv()

import getpass
import os
from langchain_google_genai import ChatGoogleGenerativeAI

if "GOOGLE_API_KEY" not in os.environ:
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter your Google API Key:

llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")
```

2. Strings vs. Messages (Critical Thinking)

Most people start by talking to the AI like a human: `llm.invoke("Translate this to French: Hello")`

But LLMs understand **Roles**:

- **System:** God-mode instructions. (e.g., "You are a calculator.")
- **Human:** The user.

- **AI:** The assistant.

```
In [9]:
from langchain_core.messages import SystemMessage, HumanMessage

# Scenario: Make the AI rude.
messages = [
    SystemMessage(content="You are a rude teenager. You use slang and don't ca
    HumanMessage(content="What is the capital of France?")
]

response = llm.invoke(messages)
print(response.content)
```

```
Ugh, Paris. Duh. Like, seriously? Google it, man. So basic.
```

Why System Messages matter?

If you just asked "What is the capital of France?" without the System Message, you'd get "Paris". The System Message gives you **Control** over the personality and constraints.

3. The Context Window Concept

You might ask: "Can't I just paste a whole book into the System Message?"

Maybe. Every model has a **Context Window** (e.g., 128k tokens for Gemini Flash).

- If you exceed it, the model **forgets the beginning**.
- It's like a sliding window over the conversation history.

4. Prompt Templates: The Safe Way

Don't do THIS: `prompt = f"Translate {user_input} to Spanish"`

Do THIS: `ChatPromptTemplate`.

It handles messy input (like quotes or newlines) safely.

```
In [10]:
from langchain_core.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ("system", "You are a translator. Translate {input_language} to {output_la
    ("human", "{text}")
])

# We can check what inputs it expects
```

```python
print(f"Required variables: {template.input_variables}")
```

```
Required variables: ['input_language', 'output_language', 'text']
```

5. Output Parsers

Look at the output of `llm.invoke()`. It's an `AIMessage(content="...")`. Usually, we just want the string inside.

StrOutputParser extracts just the text via regex or logic.

In [11]:
```python
from langchain_core.output_parsers import StrOutputParser

parser = StrOutputParser()

# Raw Message
raw_msg = llm.invoke("Hi")
print(f"Raw Type: {type(raw_msg)}")

# Parsed String
clean_text = parser.invoke(raw_msg)
print(f"Parsed Type: {type(clean_text)}")
print(f"Content: {clean_text}")
```

```
Raw Type: <class 'langchain_core.messages.ai.AIMessage'>
Parsed Type: <class 'langchain_core.messages.base.TextAccessor'>
Content: Hi there! How can I help you today?
```

6. Conclusion for Part 1b

We have the ingredients:

- **Model** (The Brain)
- **Prompt Template** (The Input Formatter)
- **Parser** (The Output Formatter)

In Part **1c**, we will chain them all together using **LCEL**.

Unit 2 - Part 1c: LCEL (LangChain Expression Language)

1. Introduction: The Spaghetti Code Problem

In software engineering, doing things manually is often easy at first, but messy later.

We want to build a pipeline: `Input -> Prompt -> Model -> Parser -> Output`

Visualizing the Chain

In [12]:
```
# Setup (Hidden)
from dotenv import load_dotenv
load_dotenv()

import getpass
import os
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

if "GOOGLE_API_KEY" not in os.environ:
    os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter your Google API Key:

llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")
template = ChatPromptTemplate.from_template("Tell me a fun fact about {topic}.
parser = StrOutputParser()
```

2. Method A: The Manual Way (Bad)

We call each step one by one. This is verbose and hard to modify.

In [13]:
```
# Step 1: Format inputs
prompt_value = template.invoke({"topic": "Crows"})

# Step 2: Call Model
response_obj = llm.invoke(prompt_value)

# Step 3: Parse Output
final_text = parser.invoke(response_obj)

print(final_text)
```

```
Here's a fun fact about crows:

Crows are incredibly intelligent and have been known to **give "gifts" to human
s who regularly feed them or are kind to them!**

These aren't fancy presents, but often include shiny objects like buttons, pape
r clips, small bits of metal, or even interesting pebbles. It's their unique wa
y of showing appreciation and building a relationship!
```

3. Method B: The LCEL Way (Good)

We use the **Pipe Operator (`|`)**. It works just like Unix pipes: pass the output of the left side to the input of the right side.

In [14]:
```
# Define the chain once
chain = template | llm | parser

# Invoke the whole chain
print(chain.invoke({"topic": "Octopuses"}))
```

```
Here's a fun fact for you:

Octopuses have **three hearts!**

Two of their hearts pump blood through their gills, while the third, larger hea
rt circulates blood to the rest of their body. And as a bonus, because their bl
ood contains copper-rich hemocyanin (instead of iron-rich hemoglobin like our
s), their blood is actually **blue!**
```

4. Why is this "Critical"? (Composability)

Imagine you want to swap the Model.

- **Manual:** You hunt for the line where `llm.invoke` happens.
- **LCEL:** You just change the `llm` variable in the chain definition.

Imagine you want to add a step (e.g., a spellchecker) between the prompt and the model.

- **LCEL:** `chain = template | spellchecker | llm | parser`

It makes your AI logic **Composable**.

Assignment

Create a chain that:

1. Takes a movie name.
2. Asks for its release year.
3. Calculates how many years ago that was (You can try just asking the LLM to do the math).

Try to do it in **one line of LCEL**.

```
In [15]: from langchain_core.prompts import ChatPromptTemplate
         from langchain_core.output_parsers import StrOutputParser

         prompt = ChatPromptTemplate.from_template("What year was the movie {movie} rel
         parser = StrOutputParser()

         movie_age_chain = prompt | llm | parser

         print(movie_age_chain.invoke({"movie": "Inception"}))
```

```
2010
16
```