LangChain: Prompt Templates (Part 1)

Prompt templates let you write a prompt once, then fill in the blanks. This makes prompts easy to reuse and easy to debug.

In LangChain v1, templates live in `langchain_core.prompts` .

We will cover three practical patterns:

- `PromptTemplate` for plain strings
- `ChatPromptTemplate` for system + human messages
- using templates directly with a chat model

Quick tip: it is always worth printing the final prompt before you call a model.

Example 1: PromptTemplate (string prompts)

We will start small with a plain string template. It outputs **one string**.

```
# Import the string prompt template class
from langchain_core.prompts import PromptTemplate
```

```
# Create a reusable string prompt template
string_prompt = PromptTemplate.from_template(
    # Placeholders live inside {braces}
    "Explain {topic} to a {audience} in exactly 2 sentences."
)
```

Now we **fill the template** and print the final string.

If the line matches, your template is working.

```
# Fill in the placeholders (LangChain calls this invoke)
string_prompt_value = string_prompt.invoke(
    # Provide values for each placeholder
    {
        "topic": "overfitting",
        "audience": "beginner"
    }
)

# Print the final formatted prompt
print(string_prompt_value.to_string())
```

```
Explain overfitting to a beginner in exactly 2 sentences.
```

Output note: You should see the filled-in prompt line. If placeholders did not fill, double-check the dictionary keys.

Example 2: ChatPromptTemplate (system + human)

Now we switch to a chat prompt. This one outputs **multiple messages**, each with a role.

```
# Import the chat prompt template class
from langchain_core.prompts import ChatPromptTemplate
```

```
# Create a chat prompt from role + message pairs
chat_prompt = ChatPromptTemplate.from_messages(
    [
        # System message sets the behavior
        ("system", "You are a helpful tutor. Answer in {format_style}."),
        # Human message is the user request
        ("human", "Teach me {topic} using one simple example")
    ]
)
```

We define a system rule and a human request, then fill the placeholders.

```
# Fill in placeholders across all chat messages
chat_prompt_value = chat_prompt.invoke(
    # Provide values for format and topic
    {
        "format_style": "two bullet points",
        "topic": "data leakage"
    }
)

# Print the final messages with their roles
for message in chat_prompt_value.to_messages():
    # Each message has a type and content
    print(f"{message.type.upper()}: {message.content}")
```

```
SYSTEM: You are a helpful tutor. Answer in two bullet points.
HUMAN: Teach me data leakage using one simple example
```

Output note: You will see SYSTEM: and HUMAN: lines - that's the exact message list the model will receive.

Initialising the model

We will reuse a lightweight chat model. (The `.env` load is just for your API key.)

```
# Import dotenv and the chat model initializer
from dotenv import load_dotenv
from langchain.chat_models import init_chat_model

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

# Create a lightweight chat model for this demo
model = init_chat_model(
    # Choose a small, fast OpenAI model
```

```
    model="gpt-4o-mini"
)
```

Example 3: Template + model

First, we send the **string prompt** to the model.

```
# Send the string prompt to the model
response = model.invoke(string_prompt_value.to_string())

# Print the assistant response
print(response.content)
```

```
Overfitting occurs when a machine learning model learns the training data too well, capturing
noise and outliers rather than the underlying patterns. As a result, the model performs poorl
y on new, unseen data because it has become too tailored to the specifics of the training se
t.
```

Output note: Expect two sentences. The wording can vary - that's normal.

Now we send the **chat prompt messages** to the model.

```
# Send the chat prompt messages to the model
response = model.invoke(chat_prompt_value.to_messages())

# Print the assistant response
print(response.content)
```

```
- **Example Scenario**: Consider a model predicting house prices based on various features li
ke square footage, number of bedrooms, and location. If the dataset includes the final sale p
rice of the houses as a feature used to train the model, this would cause data leakage becaus
e the model has access to information from the future (the actual sale price) that it would n
ot have in a real-world scenario.

- **Impact of Data Leakage**: This leads to overly optimistic model performance during traini
ng and validation, as the model learns to predict the sale price based on information it shou
ldn't have access to, ultimately resulting in poor performance on unseen data.
```

Output note: Expect two bullet points, because the system message asked for them.