**Introduction**

**LangChain** is a framework for developing applications powered by language models. It enables applications that are:

* **Data-aware**: connect a language model to other sources of data
* **Agentic**: allow a language model to interact with its environment

The main value props of LangChain are:

1. **Components**: abstractions for working with language models, along with a collection of implementations for each abstraction. Components are modular and easy-to-use, whether you are using the rest of the LangChain framework or not
2. **Off-the-shelf chains**: a structured assembly of components for accomplishing specific higher-level tasks

Off-the-shelf chains make it easy to get started. For more complex applications and nuanced use-cases, components make it easy to customize existing chains or build new ones.

**Get started**[**​**](https://python.langchain.com/docs/get_started/introduction.html#get-started)

[Here’s](https://python.langchain.com/docs/get_started/installation.html) how to install LangChain, set up your environment, and start building.

We recommend following our [Quickstart](https://python.langchain.com/docs/get_started/quickstart.html) guide to familiarize yourself with the framework by building your first LangChain application.

***Note****: These docs are for the LangChain*[*Python package*](https://github.com/hwchase17/langchain)*. For documentation on*[*LangChain.js*](https://github.com/hwchase17/langchainjs)*, the JS/TS version,*[*head here*](https://js.langchain.com/docs)*.*

**Modules**[**​**](https://python.langchain.com/docs/get_started/introduction.html#modules)

LangChain provides standard, extendable interfaces and external integrations for the following modules, listed from least to most complex:

[**Model I/O**](https://python.langchain.com/docs/modules/model_io/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#model-io)

Interface with language models

[**Data connection**](https://python.langchain.com/docs/modules/data_connection/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#data-connection)

Interface with application-specific data

[**Chains**](https://python.langchain.com/docs/modules/chains/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#chains)

Construct sequences of calls

[**Agents**](https://python.langchain.com/docs/modules/agents/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#agents)

Let chains choose which tools to use given high-level directives

[**Memory**](https://python.langchain.com/docs/modules/memory/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#memory)

Persist application state between runs of a chain

[**Callbacks**](https://python.langchain.com/docs/modules/callbacks/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#callbacks)

Log and stream intermediate steps of any chain

**Examples, ecosystem, and resources**[**​**](https://python.langchain.com/docs/get_started/introduction.html#examples-ecosystem-and-resources)

[**Use cases**](https://python.langchain.com/docs/use_cases/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#use-cases)

Walkthroughs and best-practices for common end-to-end use cases, like:

* [Chatbots](https://python.langchain.com/docs/use_cases/chatbots/)
* [Answering questions using sources](https://python.langchain.com/docs/use_cases/question_answering/)
* [Analyzing structured data](https://python.langchain.com/docs/use_cases/tabular.html)
* and much more...

[**Guides**](https://python.langchain.com/docs/guides/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#guides)

Learn best practices for developing with LangChain.

[**Ecosystem**](https://python.langchain.com/docs/ecosystem/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#ecosystem)

LangChain is part of a rich ecosystem of tools that integrate with our framework and build on top of it. Check out our growing list of [integrations](https://python.langchain.com/docs/ecosystem/integrations/) and [dependent repos](https://python.langchain.com/docs/ecosystem/dependents.html).

[**Additional resources**](https://python.langchain.com/docs/additional_resources/)[**​**](https://python.langchain.com/docs/get_started/introduction.html#additional-resources)

Our community is full of prolific developers, creative builders, and fantastic teachers. Check out [YouTube tutorials](https://python.langchain.com/docs/additional_resources/youtube.html) for great tutorials from folks in the community, and [Gallery](https://github.com/kyrolabs/awesome-langchain) for a list of awesome LangChain projects, compiled by the folks at [KyroLabs](https://kyrolabs.com/" \t "_blank).

**Support**

Join us on [GitHub](https://github.com/hwchase17/langchain) or [Discord](https://discord.gg/6adMQxSpJS) to ask questions, share feedback, meet other developers building with LangChain, and dream about the future of LLM’s.

**API reference**[**​**](https://python.langchain.com/docs/get_started/introduction.html#api-reference)

Head to the [reference](https://api.python.langchain.com/) section for full documentation of all classes and methods in the LangChain Python package.

**Installation**

**Official release**[**​**](https://python.langchain.com/docs/get_started/installation#official-release)

To install LangChain run:

* Pip
* Conda

pip install langchain

That will install the bare minimum requirements of LangChain. A lot of the value of LangChain comes when integrating it with various model providers, datastores, etc. By default, the dependencies needed to do that are NOT installed. However, there are two other ways to install LangChain that do bring in those dependencies.

To install modules needed for the common LLM providers, run:

pip install langchain[llms]

To install all modules needed for all integrations, run:

pip install langchain[all]

Note that if you are using zsh, you'll need to quote square brackets when passing them as an argument to a command, for example:

pip install 'langchain[all]'

**From source**[**​**](https://python.langchain.com/docs/get_started/installation#from-source)

If you want to install from source, you can do so by cloning the repo and running:

pip install -e .

**Quickstart**

**Installation**[**​**](https://python.langchain.com/docs/get_started/quickstart#installation)

To install LangChain run:

* Pip
* Conda

pip install langchain

For more details, see our [Installation guide](https://python.langchain.com/docs/get_started/installation.html).

**Environment setup**[**​**](https://python.langchain.com/docs/get_started/quickstart#environment-setup)

Using LangChain will usually require integrations with one or more model providers, data stores, APIs, etc. For this example, we'll use OpenAI's model APIs.

First we'll need to install their Python package:

pip install openai

Accessing the API requires an API key, which you can get by creating an account and heading [here](https://platform.openai.com/account/api-keys). Once we have a key we'll want to set it as an environment variable by running:

export OPENAI\_API\_KEY="..."

If you'd prefer not to set an environment variable you can pass the key in directly via the openai\_api\_key named parameter when initiating the OpenAI LLM class:

from langchain.llms import OpenAI  
  
llm = OpenAI(openai\_api\_key="...")

**Building an application**[**​**](https://python.langchain.com/docs/get_started/quickstart#building-an-application)

Now we can start building our language model application. LangChain provides many modules that can be used to build language model applications. Modules can be used as stand-alones in simple applications and they can be combined for more complex use cases.

**LLMs**[**​**](https://python.langchain.com/docs/get_started/quickstart#llms)

**Get predictions from a language model**[**​**](https://python.langchain.com/docs/get_started/quickstart#get-predictions-from-a-language-model)

The basic building block of LangChain is the LLM, which takes in text and generates more text.

As an example, suppose we're building an application that generates a company name based on a company description. In order to do this, we need to initialize an OpenAI model wrapper. In this case, since we want the outputs to be MORE random, we'll initialize our model with a HIGH temperature.

from langchain.llms import OpenAI  
  
llm = OpenAI(temperature=0.9)

And now we can pass in text and get predictions!

llm.predict("What would be a good company name for a company that makes colorful socks?")  
# >> Feetful of Fun

**Chat models**[**​**](https://python.langchain.com/docs/get_started/quickstart#chat-models)

Chat models are a variation on language models. While chat models use language models under the hood, the interface they expose is a bit different: rather than expose a "text in, text out" API, they expose an interface where "chat messages" are the inputs and outputs.

You can get chat completions by passing one or more messages to the chat model. The response will be a message. The types of messages currently supported in LangChain are AIMessage, HumanMessage, SystemMessage, and ChatMessage -- ChatMessage takes in an arbitrary role parameter. Most of the time, you'll just be dealing with HumanMessage, AIMessage, and SystemMessage.

from langchain.chat\_models import ChatOpenAI  
from langchain.schema import (  
 AIMessage,  
 HumanMessage,  
 SystemMessage  
)  
  
chat = ChatOpenAI(temperature=0)  
chat.predict\_messages([HumanMessage(content="Translate this sentence from English to French. I love programming.")])  
# >> AIMessage(content="J'aime programmer.", additional\_kwargs={})

It is useful to understand how chat models are different from a normal LLM, but it can often be handy to just be able to treat them the same. LangChain makes that easy by also exposing an interface through which you can interact with a chat model as you would a normal LLM. You can access this through the predict interface.

chat.predict("Translate this sentence from English to French. I love programming.")  
# >> J'aime programmer

**Prompt templates**[**​**](https://python.langchain.com/docs/get_started/quickstart#prompt-templates)

Most LLM applications do not pass user input directly into an LLM. Usually they will add the user input to a larger piece of text, called a prompt template, that provides additional context on the specific task at hand.

In the previous example, the text we passed to the model contained instructions to generate a company name. For our application, it'd be great if the user only had to provide the description of a company/product, without having to worry about giving the model instructions.

* LLMs
* Chat models

With PromptTemplates this is easy! In this case our template would be very simple:

from langchain.prompts import PromptTemplate  
  
prompt = PromptTemplate.from\_template("What is a good name for a company that makes {product}?")  
prompt.format(product="colorful socks")

What is a good name for a company that makes colorful socks?

**Chains**[**​**](https://python.langchain.com/docs/get_started/quickstart#chains)

Now that we've got a model and a prompt template, we'll want to combine the two. Chains give us a way to link (or chain) together multiple primitives, like models, prompts, and other chains.

* LLMs
* Chat models

The simplest and most common type of chain is an LLMChain, which passes an input first to a PromptTemplate and then to an LLM. We can construct an LLM chain from our existing model and prompt template.

Using this we can replace

llm.predict("What would be a good company name for a company that makes colorful socks?")

with

from langchain.chains import LLMChain  
  
chain = LLMChain(llm=llm, prompt=prompt)  
chain.run("colorful socks")

Feetful of Fun

There we go, our first chain! Understanding how this simple chain works will set you up well for working with more complex chains.

**Agents**[**​**](https://python.langchain.com/docs/get_started/quickstart#agents)

Our first chain ran a pre-determined sequence of steps. To handle complex workflows, we need to be able to dynamically choose actions based on inputs.

Agents do just this: they use a language model to determine which actions to take and in what order. Agents are given access to tools, and they repeatedly choose a tool, run the tool, and observe the output until they come up with a final answer.

To load an agent, you need to choose a(n):

* LLM/Chat model: The language model powering the agent.
* Tool(s): A function that performs a specific duty. This can be things like: Google Search, Database lookup, Python REPL, other chains. For a list of predefined tools and their specifications, see the [Tools documentation](https://python.langchain.com/docs/modules/agents/tools/).
* Agent name: A string that references a supported agent class. An agent class is largely parameterized by the prompt the language model uses to determine which action to take. Because this notebook focuses on the simplest, highest level API, this only covers using the standard supported agents. If you want to implement a custom agent, see [here](https://python.langchain.com/docs/modules/agents/how_to/custom_agent.html). For a list of supported agents and their specifications, see [here](https://python.langchain.com/docs/modules/agents/agent_types/).

For this example, we'll be using SerpAPI to query a search engine.

You'll need to install the SerpAPI Python package:

pip install google-search-results

And set the SERPAPI\_API\_KEY environment variable.

* LLMs
* Chat models

from langchain.agents import AgentType, initialize\_agent, load\_tools  
from langchain.llms import OpenAI  
  
# The language model we're going to use to control the agent.  
llm = OpenAI(temperature=0)  
  
# The tools we'll give the Agent access to. Note that the 'llm-math' tool uses an LLM, so we need to pass that in.  
tools = load\_tools(["serpapi", "llm-math"], llm=llm)  
  
# Finally, let's initialize an agent with the tools, the language model, and the type of agent we want to use.  
agent = initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION, verbose=True)  
  
# Let's test it out!  
agent.run("What was the high temperature in SF yesterday in Fahrenheit? What is that number raised to the .023 power?")

> Entering new AgentExecutor chain...  
  
Thought: I need to find the temperature first, then use the calculator to raise it to the .023 power.  
Action: Search  
Action Input: "High temperature in SF yesterday"  
Observation: San Francisco Temperature Yesterday. Maximum temperature yesterday: 57 °F (at 1:56 pm) Minimum temperature yesterday: 49 °F (at 1:56 am) Average temperature ...  
  
Thought: I now have the temperature, so I can use the calculator to raise it to the .023 power.  
Action: Calculator  
Action Input: 57^.023  
Observation: Answer: 1.0974509573251117  
  
Thought: I now know the final answer  
Final Answer: The high temperature in SF yesterday in Fahrenheit raised to the .023 power is 1.0974509573251117.  
  
> Finished chain.

The high temperature in SF yesterday in Fahrenheit raised to the .023 power is 1.0974509573251117.

**Memory**[**​**](https://python.langchain.com/docs/get_started/quickstart#memory)

The chains and agents we've looked at so far have been stateless, but for many applications it's necessary to reference past interactions. This is clearly the case with a chatbot for example, where you want it to understand new messages in the context of past messages.

The Memory module gives you a way to maintain application state. The base Memory interface is simple: it lets you update state given the latest run inputs and outputs and it lets you modify (or contextualize) the next input using the stored state.

There are a number of built-in memory systems. The simplest of these is a buffer memory which just prepends the last few inputs/outputs to the current input - we will use this in the example below.

* LLMs
* Chat models

from langchain import OpenAI, ConversationChain  
  
llm = OpenAI(temperature=0)  
conversation = ConversationChain(llm=llm, verbose=True)  
  
conversation.run("Hi there!")

here's what's going on under the hood

> Entering new chain...  
Prompt after formatting:  
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.  
  
Current conversation:  
  
Human: Hi there!  
AI:  
  
> Finished chain.  
  
>> 'Hello! How are you today?'

Now if we run the chain again

conversation.run("I'm doing well! Just having a conversation with an AI.")

we'll see that the full prompt that's passed to the model contains the input and output of our first interaction, along with our latest input

> Entering new chain...  
Prompt after formatting:  
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.  
  
Current conversation:  
  
Human: Hi there!  
AI: Hello! How are you today?  
Human: I'm doing well! Just having a conversation with an AI.  
AI:  
  
> Finished chain.  
  
>> "That's great! What would you like to talk about?"

**Modules**

LangChain provides standard, extendable interfaces and external integrations for the following modules, listed from least to most complex:

[**Model I/O**](https://python.langchain.com/docs/modules/model_io/)[**​**](https://python.langchain.com/docs/modules/#model-io)

Interface with language models

[**Data connection**](https://python.langchain.com/docs/modules/data_connection/)[**​**](https://python.langchain.com/docs/modules/#data-connection)

Interface with application-specific data

[**Chains**](https://python.langchain.com/docs/modules/chains/)[**​**](https://python.langchain.com/docs/modules/#chains)

Construct sequences of calls

[**Agents**](https://python.langchain.com/docs/modules/agents/)[**​**](https://python.langchain.com/docs/modules/#agents)

Let chains choose which tools to use given high-level directives

[**Memory**](https://python.langchain.com/docs/modules/memory/)[**​**](https://python.langchain.com/docs/modules/#memory)

Persist application state between runs of a chain

[**Callbacks**](https://python.langchain.com/docs/modules/callbacks/)[**​**](https://python.langchain.com/docs/modules/#callbacks)

Log and stream intermediate steps of any chain

[**Evaluation**](https://python.langchain.com/docs/modules/evaluation/)[**​**](https://python.langchain.com/docs/modules/#evaluation)

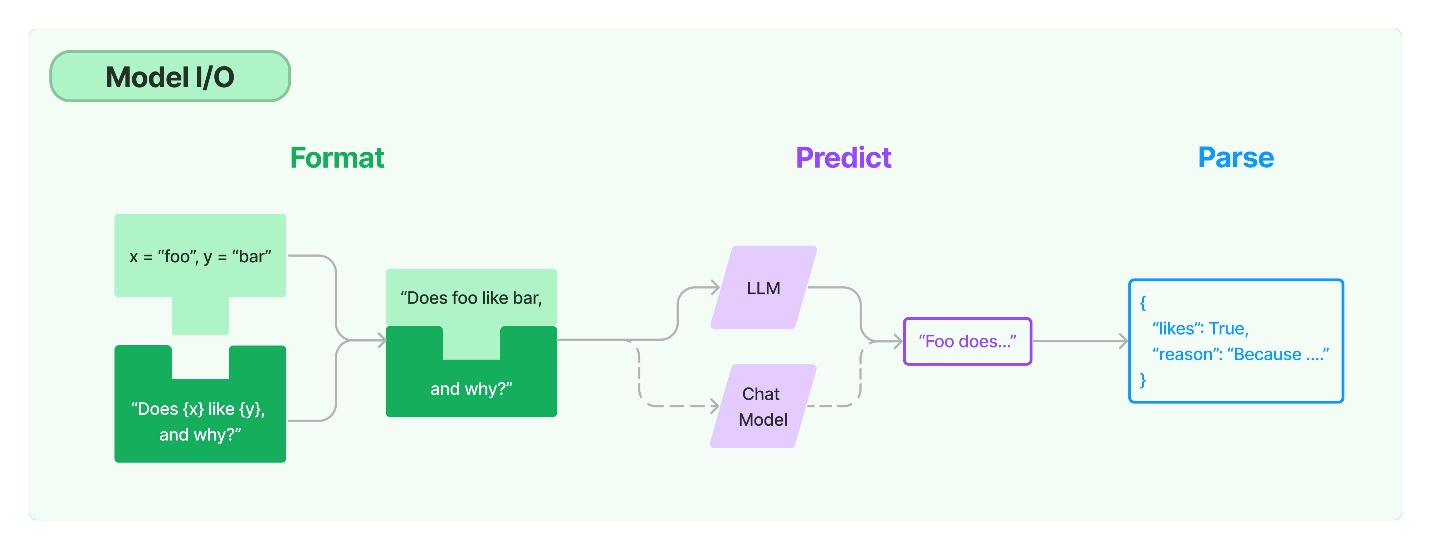
Evaluate the performance of a chain.

[Previous](https://python.langchain.com/docs/get_started/quickstart)

# Model I/O

The core element of any language model application is...the model. LangChain gives you the building blocks to interface with any language model.

* [Prompts](https://python.langchain.com/docs/modules/model_io/prompts/): Templatize, dynamically select, and manage model inputs
* [Language models](https://python.langchain.com/docs/modules/model_io/models/): Make calls to language models through common interfaces
* [Output parsers](https://python.langchain.com/docs/modules/model_io/output_parsers/): Extract information from model outputs



**Prompts**

The new way of programming models is through prompts. A **prompt** refers to the input to the model. This input is often constructed from multiple components. LangChain provides several classes and functions to make constructing and working with prompts easy.

* [Prompt templates](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/): Parametrize model inputs
* [Example selectors](https://python.langchain.com/docs/modules/model_io/prompts/example_selectors/): Dynamically select examples to include in prompts

**Prompt templates**

Language models take text as input - that text is commonly referred to as a prompt. Typically this is not simply a hardcoded string but rather a combination of a template, some examples, and user input. LangChain provides several classes and functions to make constructing and working with prompts easy.

**What is a prompt template?**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/#what-is-a-prompt-template)

A prompt template refers to a reproducible way to generate a prompt. It contains a text string ("the template"), that can take in a set of parameters from the end user and generates a prompt.

A prompt template can contain:

* instructions to the language model,
* a set of few shot examples to help the language model generate a better response,
* a question to the language model.

Here's the simplest example:

from langchain import PromptTemplate  
  
  
template = """/  
You are a naming consultant for new companies.  
What is a good name for a company that makes {product}?  
"""  
  
prompt = PromptTemplate.from\_template(template)  
prompt.format(product="colorful socks")

You are a naming consultant for new companies.  
 What is a good name for a company that makes colorful socks?

**Create a prompt template**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/#create-a-prompt-template)

You can create simple hardcoded prompts using the PromptTemplate class. Prompt templates can take any number of input variables, and can be formatted to generate a prompt.

from langchain import PromptTemplate  
  
# An example prompt with no input variables  
no\_input\_prompt = PromptTemplate(input\_variables=[], template="Tell me a joke.")  
no\_input\_prompt.format()  
# -> "Tell me a joke."  
  
# An example prompt with one input variable  
one\_input\_prompt = PromptTemplate(input\_variables=["adjective"], template="Tell me a {adjective} joke.")  
one\_input\_prompt.format(adjective="funny")  
# -> "Tell me a funny joke."  
  
# An example prompt with multiple input variables  
multiple\_input\_prompt = PromptTemplate(  
 input\_variables=["adjective", "content"],   
 template="Tell me a {adjective} joke about {content}."  
)  
multiple\_input\_prompt.format(adjective="funny", content="chickens")  
# -> "Tell me a funny joke about chickens."

If you do not wish to specify input\_variables manually, you can also create a PromptTemplate using from\_template class method. langchain will automatically infer the input\_variables based on the template passed.

template = "Tell me a {adjective} joke about {content}."  
  
prompt\_template = PromptTemplate.from\_template(template)  
prompt\_template.input\_variables  
# -> ['adjective', 'content']  
prompt\_template.format(adjective="funny", content="chickens")  
# -> Tell me a funny joke about chickens.

You can create custom prompt templates that format the prompt in any way you want. For more information, see [Custom Prompt Templates](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/custom_prompt_template.html).

**Chat prompt template**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/#chat-prompt-template)

[Chat Models](https://python.langchain.com/docs/modules/model_io/prompts/models/chat) take a list of chat messages as input - this list commonly referred to as a prompt. These chat messages differ from raw string (which you would pass into a [LLM](https://python.langchain.com/docs/modules/model_io/models/llms) model) in that every message is associated with a role.

For example, in OpenAI [Chat Completion API](https://platform.openai.com/docs/guides/chat/introduction), a chat message can be associated with the AI, human or system role. The model is supposed to follow instruction from system chat message more closely.

LangChain provides several prompt templates to make constructing and working with prompts easily. You are encouraged to use these chat related prompt templates instead of PromptTemplate when querying chat models to fully exploit the potential of underlying chat model.

from langchain.prompts import (  
 ChatPromptTemplate,  
 PromptTemplate,  
 SystemMessagePromptTemplate,  
 AIMessagePromptTemplate,  
 HumanMessagePromptTemplate,  
)  
from langchain.schema import (  
 AIMessage,  
 HumanMessage,  
 SystemMessage  
)

To create a message template associated with a role, you use MessagePromptTemplate.

For convenience, there is a from\_template method exposed on the template. If you were to use this template, this is what it would look like:

template="You are a helpful assistant that translates {input\_language} to {output\_language}."  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)  
human\_template="{text}"  
human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)

If you wanted to construct the MessagePromptTemplate more directly, you could create a PromptTemplate outside and then pass it in, eg:

prompt=PromptTemplate(  
 template="You are a helpful assistant that translates {input\_language} to {output\_language}.",  
 input\_variables=["input\_language", "output\_language"],  
)  
system\_message\_prompt\_2 = SystemMessagePromptTemplate(prompt=prompt)  
  
assert system\_message\_prompt == system\_message\_prompt\_2

After that, you can build a ChatPromptTemplate from one or more MessagePromptTemplates. You can use ChatPromptTemplate's format\_prompt -- this returns a PromptValue, which you can convert to a string or Message object, depending on whether you want to use the formatted value as input to an llm or chat model.

chat\_prompt = ChatPromptTemplate.from\_messages([system\_message\_prompt, human\_message\_prompt])  
  
# get a chat completion from the formatted messages  
chat\_prompt.format\_prompt(input\_language="English", output\_language="French", text="I love programming.").to\_messages()

[SystemMessage(content='You are a helpful assistant that translates English to French.', additional\_kwargs={}),  
 HumanMessage(content='I love programming.', additional\_kwargs={})]

**Connecting to a Feature Store**

Feature stores are a concept from traditional machine learning that make sure data fed into models is up-to-date and relevant. For more on this, see [here](https://www.tecton.ai/blog/what-is-a-feature-store/).

This concept is extremely relevant when considering putting LLM applications in production. In order to personalize LLM applications, you may want to combine LLMs with up-to-date information about particular users. Feature stores can be a great way to keep that data fresh, and LangChain provides an easy way to combine that data with LLMs.

In this notebook we will show how to connect prompt templates to feature stores. The basic idea is to call a feature store from inside a prompt template to retrieve values that are then formatted into the prompt.

**Feast**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#feast)

To start, we will use the popular open source feature store framework [Feast](https://github.com/feast-dev/feast).

This assumes you have already run the steps in the README around getting started. We will build of off that example in getting started, and create and LLMChain to write a note to a specific driver regarding their up-to-date statistics.

**Load Feast Store**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#load-feast-store)

Again, this should be set up according to the instructions in the Feast README

from feast import FeatureStore  
  
# You may need to update the path depending on where you stored it  
feast\_repo\_path = "../../../../../my\_feature\_repo/feature\_repo/"  
store = FeatureStore(repo\_path=feast\_repo\_path)

**Prompts**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#prompts)

Here we will set up a custom FeastPromptTemplate. This prompt template will take in a driver id, look up their stats, and format those stats into a prompt.

Note that the input to this prompt template is just driver\_id, since that is the only user defined piece (all other variables are looked up inside the prompt template).

from langchain.prompts import PromptTemplate, StringPromptTemplate

template = """Given the driver's up to date stats, write them note relaying those stats to them.  
If they have a conversation rate above .5, give them a compliment. Otherwise, make a silly joke about chickens at the end to make them feel better  
  
Here are the drivers stats:  
Conversation rate: {conv\_rate}  
Acceptance rate: {acc\_rate}  
Average Daily Trips: {avg\_daily\_trips}  
  
Your response:"""  
prompt = PromptTemplate.from\_template(template)

class FeastPromptTemplate(StringPromptTemplate):  
 def format(self, \*\*kwargs) -> str:  
 driver\_id = kwargs.pop("driver\_id")  
 feature\_vector = store.get\_online\_features(  
 features=[  
 "driver\_hourly\_stats:conv\_rate",  
 "driver\_hourly\_stats:acc\_rate",  
 "driver\_hourly\_stats:avg\_daily\_trips",  
 ],  
 entity\_rows=[{"driver\_id": driver\_id}],  
 ).to\_dict()  
 kwargs["conv\_rate"] = feature\_vector["conv\_rate"][0]  
 kwargs["acc\_rate"] = feature\_vector["acc\_rate"][0]  
 kwargs["avg\_daily\_trips"] = feature\_vector["avg\_daily\_trips"][0]  
 return prompt.format(\*\*kwargs)

prompt\_template = FeastPromptTemplate(input\_variables=["driver\_id"])

print(prompt\_template.format(driver\_id=1001))

Given the driver's up to date stats, write them note relaying those stats to them.  
 If they have a conversation rate above .5, give them a compliment. Otherwise, make a silly joke about chickens at the end to make them feel better  
   
 Here are the drivers stats:  
 Conversation rate: 0.4745151400566101  
 Acceptance rate: 0.055561766028404236  
 Average Daily Trips: 936  
   
 Your response:

**Use in a chain**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#use-in-a-chain)

We can now use this in a chain, successfully creating a chain that achieves personalization backed by a feature store

from langchain.chat\_models import ChatOpenAI  
from langchain.chains import LLMChain

chain = LLMChain(llm=ChatOpenAI(), prompt=prompt\_template)

chain.run(1001)

"Hi there! I wanted to update you on your current stats. Your acceptance rate is 0.055561766028404236 and your average daily trips are 936. While your conversation rate is currently 0.4745151400566101, I have no doubt that with a little extra effort, you'll be able to exceed that .5 mark! Keep up the great work! And remember, even chickens can't always cross the road, but they still give it their best shot."

**Tecton[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store" \l "tecton" \o "Direct link to Tecton)**

Above, we showed how you could use Feast, a popular open source and self-managed feature store, with LangChain. Our examples below will show a similar integration using Tecton. Tecton is a fully managed feature platform built to orchestrate the complete ML feature lifecycle, from transformation to online serving, with enterprise-grade SLAs.

**Prerequisites**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#prerequisites)

* Tecton Deployment (sign up at [https://tecton.ai](https://tecton.ai/))
* TECTON\_API\_KEY environment variable set to a valid Service Account key

**Define and Load Features**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#define-and-load-features)

We will use the user\_transaction\_counts Feature View from the [Tecton tutorial](https://docs.tecton.ai/docs/tutorials/tecton-fundamentals" \t "_blank) as part of a Feature Service. For simplicity, we are only using a single Feature View; however, more sophisticated applications may require more feature views to retrieve the features needed for its prompt.

user\_transaction\_metrics = FeatureService(  
 name = "user\_transaction\_metrics",  
 features = [user\_transaction\_counts]  
)

The above Feature Service is expected to be [applied to a live workspace](https://docs.tecton.ai/docs/applying-feature-repository-changes-to-a-workspace). For this example, we will be using the "prod" workspace.

import tecton  
  
workspace = tecton.get\_workspace("prod")  
feature\_service = workspace.get\_feature\_service("user\_transaction\_metrics")

**Prompts**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#prompts-1)

Here we will set up a custom TectonPromptTemplate. This prompt template will take in a user\_id , look up their stats, and format those stats into a prompt.

Note that the input to this prompt template is just user\_id, since that is the only user defined piece (all other variables are looked up inside the prompt template).

from langchain.prompts import PromptTemplate, StringPromptTemplate

template = """Given the vendor's up to date transaction stats, write them a note based on the following rules:  
  
1. If they had a transaction in the last day, write a short congratulations message on their recent sales  
2. If no transaction in the last day, but they had a transaction in the last 30 days, playfully encourage them to sell more.  
3. Always add a silly joke about chickens at the end  
  
Here are the vendor's stats:  
Number of Transactions Last Day: {transaction\_count\_1d}  
Number of Transactions Last 30 Days: {transaction\_count\_30d}  
  
Your response:"""  
prompt = PromptTemplate.from\_template(template)

class TectonPromptTemplate(StringPromptTemplate):  
 def format(self, \*\*kwargs) -> str:  
 user\_id = kwargs.pop("user\_id")  
 feature\_vector = feature\_service.get\_online\_features(  
 join\_keys={"user\_id": user\_id}  
 ).to\_dict()  
 kwargs["transaction\_count\_1d"] = feature\_vector[  
 "user\_transaction\_counts.transaction\_count\_1d\_1d"  
 ]  
 kwargs["transaction\_count\_30d"] = feature\_vector[  
 "user\_transaction\_counts.transaction\_count\_30d\_1d"  
 ]  
 return prompt.format(\*\*kwargs)

prompt\_template = TectonPromptTemplate(input\_variables=["user\_id"])

print(prompt\_template.format(user\_id="user\_469998441571"))

Given the vendor's up to date transaction stats, write them a note based on the following rules:  
   
 1. If they had a transaction in the last day, write a short congratulations message on their recent sales  
 2. If no transaction in the last day, but they had a transaction in the last 30 days, playfully encourage them to sell more.  
 3. Always add a silly joke about chickens at the end  
   
 Here are the vendor's stats:  
 Number of Transactions Last Day: 657  
 Number of Transactions Last 30 Days: 20326  
   
 Your response:

**Use in a chain**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#use-in-a-chain-1)

We can now use this in a chain, successfully creating a chain that achieves personalization backed by the Tecton Feature Platform

from langchain.chat\_models import ChatOpenAI  
from langchain.chains import LLMChain

chain = LLMChain(llm=ChatOpenAI(), prompt=prompt\_template)

chain.run("user\_469998441571")

'Wow, congratulations on your recent sales! Your business is really soaring like a chicken on a hot air balloon! Keep up the great work!'

**Featureform[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store" \l "featureform" \o "Direct link to Featureform)**

Finally, we will use [Featureform](https://github.com/featureform/featureform" \t "_blank) an open-source and enterprise-grade feature store to run the same example. Featureform allows you to work with your infrastructure like Spark or locally to define your feature transformations.

**Initialize Featureform[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store" \l "initialize-featureform" \o "Direct link to Initialize Featureform)**

You can follow in the instructions in the README to initialize your transformations and features in Featureform.

import featureform as ff  
  
client = ff.Client(host="demo.featureform.com")

**Prompts**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#prompts-2)

Here we will set up a custom FeatureformPromptTemplate. This prompt template will take in the average amount a user pays per transactions.

Note that the input to this prompt template is just avg\_transaction, since that is the only user defined piece (all other variables are looked up inside the prompt template).

from langchain.prompts import PromptTemplate, StringPromptTemplate

template = """Given the amount a user spends on average per transaction, let them know if they are a high roller. Otherwise, make a silly joke about chickens at the end to make them feel better  
  
Here are the user's stats:  
Average Amount per Transaction: ${avg\_transcation}  
  
Your response:"""  
prompt = PromptTemplate.from\_template(template)

class FeatureformPromptTemplate(StringPromptTemplate):  
 def format(self, \*\*kwargs) -> str:  
 user\_id = kwargs.pop("user\_id")  
 fpf = client.features([("avg\_transactions", "quickstart")], {"user": user\_id})  
 return prompt.format(\*\*kwargs)

prompt\_template = FeatureformPrompTemplate(input\_variables=["user\_id"])

print(prompt\_template.format(user\_id="C1410926"))

**Use in a chain**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/connecting_to_a_feature_store#use-in-a-chain-2)

We can now use this in a chain, successfully creating a chain that achieves personalization backed by the Featureform Feature Platform

from langchain.chat\_models import ChatOpenAI  
from langchain.chains import LLMChain

chain = LLMChain(llm=ChatOpenAI(), prompt=prompt\_template)

chain.run("C1410926")

**Custom prompt template**

Let's suppose we want the LLM to generate English language explanations of a function given its name. To achieve this task, we will create a custom prompt template that takes in the function name as input, and formats the prompt template to provide the source code of the function.

**Why are custom prompt templates needed?**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/custom_prompt_template#why-are-custom-prompt-templates-needed)

LangChain provides a set of default prompt templates that can be used to generate prompts for a variety of tasks. However, there may be cases where the default prompt templates do not meet your needs. For example, you may want to create a prompt template with specific dynamic instructions for your language model. In such cases, you can create a custom prompt template.

Take a look at the current set of default prompt templates [here](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/).

**Creating a Custom Prompt Template**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/custom_prompt_template#creating-a-custom-prompt-template)

There are essentially two distinct prompt templates available - string prompt templates and chat prompt templates. String prompt templates provides a simple prompt in string format, while chat prompt templates produces a more structured prompt to be used with a chat API.

In this guide, we will create a custom prompt using a string prompt template.

To create a custom string prompt template, there are two requirements:

1. It has an input\_variables attribute that exposes what input variables the prompt template expects.
2. It exposes a format method that takes in keyword arguments corresponding to the expected input\_variables and returns the formatted prompt.

We will create a custom prompt template that takes in the function name as input and formats the prompt to provide the source code of the function. To achieve this, let's first create a function that will return the source code of a function given its name.

import inspect  
  
  
def get\_source\_code(function\_name):  
 # Get the source code of the function  
 return inspect.getsource(function\_name)

Next, we'll create a custom prompt template that takes in the function name as input, and formats the prompt template to provide the source code of the function.

from langchain.prompts import StringPromptTemplate  
from pydantic import BaseModel, validator  
  
  
class FunctionExplainerPromptTemplate(StringPromptTemplate, BaseModel):  
 """A custom prompt template that takes in the function name as input, and formats the prompt template to provide the source code of the function."""  
  
 @validator("input\_variables")  
 def validate\_input\_variables(cls, v):  
 """Validate that the input variables are correct."""  
 if len(v) != 1 or "function\_name" not in v:  
 raise ValueError("function\_name must be the only input\_variable.")  
 return v  
  
 def format(self, \*\*kwargs) -> str:  
 # Get the source code of the function  
 source\_code = get\_source\_code(kwargs["function\_name"])  
  
 # Generate the prompt to be sent to the language model  
 prompt = f"""  
 Given the function name and source code, generate an English language explanation of the function.  
 Function Name: {kwargs["function\_name"].\_\_name\_\_}  
 Source Code:  
 {source\_code}  
 Explanation:  
 """  
 return prompt  
  
 def \_prompt\_type(self):  
 return "function-explainer"

**Use the custom prompt template**[**​**](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/custom_prompt_template#use-the-custom-prompt-template)

Now that we have created a custom prompt template, we can use it to generate prompts for our task.

fn\_explainer = FunctionExplainerPromptTemplate(input\_variables=["function\_name"])  
  
# Generate a prompt for the function "get\_source\_code"  
prompt = fn\_explainer.format(function\_name=get\_source\_code)  
print(prompt)

Given the function name and source code, generate an English language explanation of the function.  
 Function Name: get\_source\_code  
 Source Code:  
 def get\_source\_code(function\_name):  
 # Get the source code of the function  
 return inspect.getsource(function\_name)  
   
 Explanation:

# Few-shot prompt templates

In this tutorial, we'll learn how to create a prompt template that uses few shot examples. A few shot prompt template can be constructed from either a set of examples, or from an Example Selector object.

### Use Case[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples#use-case)

In this tutorial, we'll configure few shot examples for self-ask with search.

## Using an example set[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples#using-an-example-set)

### Create the example set[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples#create-the-example-set)

To get started, create a list of few shot examples. Each example should be a dictionary with the keys being the input variables and the values being the values for those input variables.

from langchain.prompts.few\_shot import FewShotPromptTemplate  
from langchain.prompts.prompt import PromptTemplate  
  
examples = [  
 {  
 "question": "Who lived longer, Muhammad Ali or Alan Turing?",  
 "answer":   
"""  
Are follow up questions needed here: Yes.  
Follow up: How old was Muhammad Ali when he died?  
Intermediate answer: Muhammad Ali was 74 years old when he died.  
Follow up: How old was Alan Turing when he died?  
Intermediate answer: Alan Turing was 41 years old when he died.  
So the final answer is: Muhammad Ali  
"""  
 },  
 {  
 "question": "When was the founder of craigslist born?",  
 "answer":   
"""  
Are follow up questions needed here: Yes.  
Follow up: Who was the founder of craigslist?  
Intermediate answer: Craigslist was founded by Craig Newmark.  
Follow up: When was Craig Newmark born?  
Intermediate answer: Craig Newmark was born on December 6, 1952.  
So the final answer is: December 6, 1952  
"""  
 },  
 {  
 "question": "Who was the maternal grandfather of George Washington?",  
 "answer":  
"""  
Are follow up questions needed here: Yes.  
Follow up: Who was the mother of George Washington?  
Intermediate answer: The mother of George Washington was Mary Ball Washington.  
Follow up: Who was the father of Mary Ball Washington?  
Intermediate answer: The father of Mary Ball Washington was Joseph Ball.  
So the final answer is: Joseph Ball  
"""  
 },  
 {  
 "question": "Are both the directors of Jaws and Casino Royale from the same country?",  
 "answer":  
"""  
Are follow up questions needed here: Yes.  
Follow up: Who is the director of Jaws?  
Intermediate Answer: The director of Jaws is Steven Spielberg.  
Follow up: Where is Steven Spielberg from?  
Intermediate Answer: The United States.  
Follow up: Who is the director of Casino Royale?  
Intermediate Answer: The director of Casino Royale is Martin Campbell.  
Follow up: Where is Martin Campbell from?  
Intermediate Answer: New Zealand.  
So the final answer is: No  
"""  
 }  
]

### Create a formatter for the few shot examples[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples#create-a-formatter-for-the-few-shot-examples)

Configure a formatter that will format the few shot examples into a string. This formatter should be a PromptTemplate object.

example\_prompt = PromptTemplate(input\_variables=["question", "answer"], template="Question: {question}\n{answer}")  
  
print(example\_prompt.format(\*\*examples[0]))

Question: Who lived longer, Muhammad Ali or Alan Turing?  
   
 Are follow up questions needed here: Yes.  
 Follow up: How old was Muhammad Ali when he died?  
 Intermediate answer: Muhammad Ali was 74 years old when he died.  
 Follow up: How old was Alan Turing when he died?  
 Intermediate answer: Alan Turing was 41 years old when he died.  
 So the final answer is: Muhammad Ali

### Feed examples and formatter to FewShotPromptTemplate[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples" \l "feed-examples-and-formatter-to-fewshotprompttemplate" \o "Direct link to feed-examples-and-formatter-to-fewshotprompttemplate)

Finally, create a FewShotPromptTemplate object. This object takes in the few shot examples and the formatter for the few shot examples.

prompt = FewShotPromptTemplate(  
 examples=examples,   
 example\_prompt=example\_prompt,   
 suffix="Question: {input}",   
 input\_variables=["input"]  
)  
  
print(prompt.format(input="Who was the father of Mary Ball Washington?"))

Question: Who lived longer, Muhammad Ali or Alan Turing?  
   
 Are follow up questions needed here: Yes.  
 Follow up: How old was Muhammad Ali when he died?  
 Intermediate answer: Muhammad Ali was 74 years old when he died.  
 Follow up: How old was Alan Turing when he died?  
 Intermediate answer: Alan Turing was 41 years old when he died.  
 So the final answer is: Muhammad Ali  
   
   
 Question: When was the founder of craigslist born?  
   
 Are follow up questions needed here: Yes.  
 Follow up: Who was the founder of craigslist?  
 Intermediate answer: Craigslist was founded by Craig Newmark.  
 Follow up: When was Craig Newmark born?  
 Intermediate answer: Craig Newmark was born on December 6, 1952.  
 So the final answer is: December 6, 1952  
   
   
 Question: Who was the maternal grandfather of George Washington?  
   
 Are follow up questions needed here: Yes.  
 Follow up: Who was the mother of George Washington?  
 Intermediate answer: The mother of George Washington was Mary Ball Washington.  
 Follow up: Who was the father of Mary Ball Washington?  
 Intermediate answer: The father of Mary Ball Washington was Joseph Ball.  
 So the final answer is: Joseph Ball  
   
   
 Question: Are both the directors of Jaws and Casino Royale from the same country?  
   
 Are follow up questions needed here: Yes.  
 Follow up: Who is the director of Jaws?  
 Intermediate Answer: The director of Jaws is Steven Spielberg.  
 Follow up: Where is Steven Spielberg from?  
 Intermediate Answer: The United States.  
 Follow up: Who is the director of Casino Royale?  
 Intermediate Answer: The director of Casino Royale is Martin Campbell.  
 Follow up: Where is Martin Campbell from?  
 Intermediate Answer: New Zealand.  
 So the final answer is: No  
   
   
 Question: Who was the father of Mary Ball Washington?

## Using an example selector[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples#using-an-example-selector)

### Feed examples into ExampleSelector[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples" \l "feed-examples-into-exampleselector" \o "Direct link to feed-examples-into-exampleselector)

We will reuse the example set and the formatter from the previous section. However, instead of feeding the examples directly into the FewShotPromptTemplate object, we will feed them into an ExampleSelector object.

In this tutorial, we will use the SemanticSimilarityExampleSelector class. This class selects few shot examples based on their similarity to the input. It uses an embedding model to compute the similarity between the input and the few shot examples, as well as a vector store to perform the nearest neighbor search.

from langchain.prompts.example\_selector import SemanticSimilarityExampleSelector  
from langchain.vectorstores import Chroma  
from langchain.embeddings import OpenAIEmbeddings  
  
  
example\_selector = SemanticSimilarityExampleSelector.from\_examples(  
 # This is the list of examples available to select from.  
 examples,  
 # This is the embedding class used to produce embeddings which are used to measure semantic similarity.  
 OpenAIEmbeddings(),  
 # This is the VectorStore class that is used to store the embeddings and do a similarity search over.  
 Chroma,  
 # This is the number of examples to produce.  
 k=1  
)  
  
# Select the most similar example to the input.  
question = "Who was the father of Mary Ball Washington?"  
selected\_examples = example\_selector.select\_examples({"question": question})  
print(f"Examples most similar to the input: {question}")  
for example in selected\_examples:  
 print("\n")  
 for k, v in example.items():  
 print(f"{k}: {v}")

Running Chroma using direct local API.  
 Using DuckDB in-memory for database. Data will be transient.  
 Examples most similar to the input: Who was the father of Mary Ball Washington?  
   
   
 question: Who was the maternal grandfather of George Washington?  
 answer:   
 Are follow up questions needed here: Yes.  
 Follow up: Who was the mother of George Washington?  
 Intermediate answer: The mother of George Washington was Mary Ball Washington.  
 Follow up: Who was the father of Mary Ball Washington?  
 Intermediate answer: The father of Mary Ball Washington was Joseph Ball.  
 So the final answer is: Joseph Ball

### Feed example selector into FewShotPromptTemplate[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples" \l "feed-example-selector-into-fewshotprompttemplate" \o "Direct link to feed-example-selector-into-fewshotprompttemplate)

Finally, create a FewShotPromptTemplate object. This object takes in the example selector and the formatter for the few shot examples.

prompt = FewShotPromptTemplate(  
 example\_selector=example\_selector,   
 example\_prompt=example\_prompt,   
 suffix="Question: {input}",   
 input\_variables=["input"]  
)  
  
print(prompt.format(input="Who was the father of Mary Ball Washington?"))

Question: Who was the maternal grandfather of George Washington?  
   
 Are follow up questions needed here: Yes.  
 Follow up: Who was the mother of George Washington?  
 Intermediate answer: The mother of George Washington was Mary Ball Washington.  
 Follow up: Who was the father of Mary Ball Washington?  
 Intermediate answer: The father of Mary Ball Washington was Joseph Ball.  
 So the final answer is: Joseph Ball  
   
   
 Question: Who was the father of Mary Ball Washington?

# Few shot examples for chat models

This notebook covers how to use few shot examples in chat models.

There does not appear to be solid consensus on how best to do few shot prompting. As a result, we are not solidifying any abstractions around this yet but rather using existing abstractions.

## Alternating Human/AI messages[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples_chat#alternating-humanai-messages)

The first way of doing few shot prompting relies on using alternating human/ai messages. See an example of this below.

from langchain.chat\_models import ChatOpenAI  
from langchain import PromptTemplate, LLMChain  
from langchain.prompts.chat import (  
 ChatPromptTemplate,  
 SystemMessagePromptTemplate,  
 AIMessagePromptTemplate,  
 HumanMessagePromptTemplate,  
)  
from langchain.schema import AIMessage, HumanMessage, SystemMessage

chat = ChatOpenAI(temperature=0)

template = "You are a helpful assistant that translates english to pirate."  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)  
example\_human = HumanMessagePromptTemplate.from\_template("Hi")  
example\_ai = AIMessagePromptTemplate.from\_template("Argh me mateys")  
human\_template = "{text}"  
human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)

chat\_prompt = ChatPromptTemplate.from\_messages(  
 [system\_message\_prompt, example\_human, example\_ai, human\_message\_prompt]  
)  
chain = LLMChain(llm=chat, prompt=chat\_prompt)  
# get a chat completion from the formatted messages  
chain.run("I love programming.")

"I be lovin' programmin', me hearty!"

## System Messages[​](https://python.langchain.com/docs/modules/model_io/prompts/prompt_templates/few_shot_examples_chat#system-messages)

OpenAI provides an optional name parameter that they also recommend using in conjunction with system messages to do few shot prompting. Here is an example of how to do that below.

template = "You are a helpful assistant that translates english to pirate."  
system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)  
example\_human = SystemMessagePromptTemplate.from\_template(  
 "Hi", additional\_kwargs={"name": "example\_user"}  
)  
example\_ai = SystemMessagePromptTemplate.from\_template(  
 "Argh me mateys", additional\_kwargs={"name": "example\_assistant"}  
)  
human\_template = "{text}"  
human\_message\_prompt = HumanMessagePromptTemplate.from\_template(human\_template)

chat\_prompt = ChatPromptTemplate.from\_messages(  
 [system\_message\_prompt, example\_human, example\_ai, human\_message\_prompt]  
)  
chain = LLMChain(llm=chat, prompt=chat\_prompt)  
# get a chat completion from the formatted messages  
chain.run("I love programming.")

"I be lovin' programmin', me hearty."