Quickstart Guide

Contents

This tutorial gives you a quick walkthrough about building an end-to-end language model application with LangChain.

Installation

To get started, install LangChain with the following command:

```
pip install langchain
# or
conda install langchain -c conda-forge
```

Environment Setup

Using LangChain will usually require integrations with one or more model providers, data stores, apis, etc.
For this example, we will be using OpenAI's APIs, so we will first need to install their SDK:

```
pip install openai
```

We will then need to set the environment variable in the terminal.

```
export OPENAI_API_KEY="..."
```

Alternatively, you could do this from inside the Jupyter notebook (or Python script):

```
import os
os.environ["OPENAI_API_KEY"] = "..."
```

Building a Language Model Application: LLMs

Now that we have installed LangChain and set up our environment, we can start building our language model application.
LangChain provides many modules that can be used to build language model applications. Modules can be combined to create more complex applications, or be used individually for simple applications.

LLMs: Get predictions from a language model

The most basic building block of LangChain is calling an LLM on some input. Let's walk through a simple example of how to do this. For this purpose, let's pretend we are building a service that generates a company name based on what the company makes.
In order to do this, we first need to import the LLM wrapper.

```
from langchain.llms import OpenAI
```

We can then initialize the wrapper with any arguments. In this example, we probably want the outputs to be MORE random, so we'll initialize it with a HIGH temperature.

```
llm = OpenAI(temperature=0.9)
```

We can now call it on some input!

```
text = "What would be a good company name for a company that makes colorful socks?"
print(llm(text))
```

```
Feetful of Fun
```

For more details on how to use LLMs within LangChain, see the LLM getting started guide.

Prompt Templates: Manage prompts for LLMs

Calling an LLM is a great first step, but it's just the beginning. Normally when you use an LLM in an application, you are not sending user input directly to the LLM. Instead, you are probably taking user input and constructing a prompt, and then sending that to the LLM.
For example, in the previous example, the text we passed in was hardcoded to ask for a name for a company that made colorful socks. In this imaginary service, what we would want to do is take only the user input describing what the company does, and then format the prompt with that information.
This is easy to do with LangChain!
First lets define the prompt template:

```
from langchain.prompts import PromptTemplate

prompt = PromptTemplate(
    input_variables=["product"],
    template="What is a good name for a company that makes {product}?",
)
```

```
print(prompt.format(product="colorful socks"))
```

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

For more details, check out the getting started guide for prompts.

Chains: Combine LLMs and prompts in multi-step workflows

Up until now, we've worked with the PromptTemplate and LLM primitives by themselves. But of course, a real application is not just one primitive, but rather a combination of them.
A chain in LangChain is made up of links, which can be either primitives like LLMs or other chains.
The most core type of chain is an LLMChain, which consists of a PromptTemplate and an LLM.
Extending the previous example, we can construct an LLMChain which takes user input, formats it with a PromptTemplate, and then passes the formatted response to an LLM.

```
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI

llm = OpenAI(temperature=0.9)
prompt = PromptTemplate(
    input_variables=["product"],
    template="What is a good name for a company that makes {product}?",
)
```

We can now create a very simple chain that will take user input, format the prompt with it, and then send it to the LLM:

```
from langchain.chains import LLMChain
chain = LLMChain(llm=llm, prompt=prompt)
```

Now we can run that chain only specifying the product!

```
chain.run("colorful socks")
```


Skip to main content

There we go! There's the first chain - an LLM Chain. This is one of the simpler types of chains, but understanding how it works will set you up well for working with more complex chains.
For more details, check out the getting started guide for chains.

Agents: Dynamically Call Chains Based on User Input

So far the chains we've looked at run in a predetermined order.
Agents no longer do: they use an LLM to determine which actions to take and in what order.
An action can either be using a tool and observing its output, or returning to the user.
When used correctly agents can be extremely powerful. In this tutorial, we show you how to easily use agents through the simplest, highest level API.
In order to load agents, you should understand the following concepts:

- Tool: A function that performs a specific duty. This can be things like: Google Search, Database lookup, Python REPL, other chains. The interface for a tool is currently a function that is expected to have a string as an input, with a string as an output.
- LLM: The language model powering the agent.
- Agent: The agent to use. This should be a string that references a support agent class. 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 the documentation for custom agents (coming soon).

Agents: For a list of supported agents and their specifications, see here.
Tools: For a list of predefined tools and their specifications, see here.
For this example, you will also need to install the SerpAPI Python package.

```
pip install google-search-results
```

And set the appropriate environment variables.

```
import os
os.environ["SERPAPI_API_KEY"] = "..."
```

Now we can get started!

Skip to main content

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import OpenAI

# First, let's load the language model we're going to use to control the
agent.
llm = OpenAI(temperature=0)

# Next, let's load some tools to use. 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)

# Now 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...
 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.
```

Memory: Add State to Chains and Agents

So far, all the chains and agents we've gone through have been stateless. But ofter y

want a chain or agent to have some concept of "memory" so that it may remember

designing a chatbot - you want it to remember previous messages so it can use context from that to have a better conversation. This would be a type of “short-term memory”. On the more complex side, you could imagine a chain/agent remembering key pieces of information over time - this would be a form of “long-term memory”. For more concrete ideas on the latter, see this awesome paper.

LangChain provides several specially created chains just for this purpose. This notebook walks through using one of those chains (the `ConversationChain`) with two different types of memory.

By default, the `ConversationChain` has a simple type of memory that remembers all previous inputs/outputs and adds them to the context that is passed. Let’s take a look at using this chain (setting `verbose=True` so we can see the prompt).

```
from langchain import OpenAI, ConversationChain

llm = OpenAI(temperature=0)
conversation = ConversationChain(llm=llm, verbose=True)

output = conversation.predict(input="Hi there!")
print(output)
```

```
> 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?'
```

```
output = conversation.predict(input="I'm doing well! Just having a
conversation with an AI.")
print(output)
```

```
> Entering new chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. T
```


Skip to main content

```
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?"
```

Building a Language Model Application: Chat Models

Similarly, you can use chat models instead of LLMs. 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.
Chat model APIs are fairly new, so we are still figuring out the correct abstractions.

Get Message Completions from a Chat Model

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)
```

You can get completions by passing in a single message.

```
chat([HumanMessage(content="Translate this sentence from English to French.
I love programming.")])
# -> AIMessage(content="J'aime programmer.", additional_kwargs={})
```

You can also pass in multiple messages for OpenAI's gpt-3.5-turbo and gpt-4 models.

```
messages = [
    SystemMessage(content="You are a helpful assistant that translates
English to French."),
    HumanMessage(content="I love programming.")
]
chat(messages)
# -> AIMessage(content="J'aime programmer.", additional_kwargs={})
```

You can go one step further and generate completions for multiple sets of messages using `generate`. This returns an `LLMResult` with an additional `message` parameter:

```
batch_messages = [
    [
        SystemMessage(content="You are a helpful assistant that translates
English to French."),
        HumanMessage(content="I love programming.")
    ],
    [
        SystemMessage(content="You are a helpful assistant that translates
English to French."),
        HumanMessage(content="I love artificial intelligence.")
    ],
]
result = chat.generate(batch_messages)
result
# -> LLMResult(generations=[[ChatGeneration(text="J'aime programmer.",
generation_info=None, message=AIMessage(content="J'aime programmer.",
additional_kwargs={}))], [ChatGeneration(text="J'aime l'intelligence
artificielle.", generation_info=None, message=AIMessage(content="J'aime
l'intelligence artificielle.", additional_kwargs={}))]], llm_output=
{'token_usage': {'prompt_tokens': 57, 'completion_tokens': 20,
'total_tokens': 77}})
```

You can recover things like token usage from this LLMResult:

```
result.llm_output['token_usage']
# -> {'prompt_tokens': 57, 'completion_tokens': 20, 'total_tokens': 7
```

Chat Prompt Templates

Similar to LLMs, you can make use of templating by using a `MessagePromptTemplate`. You can build a `ChatPromptTemplate` from one or more `MessagePromptTemplate`s. 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.

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:

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

chat = ChatOpenAI(temperature=0)

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)

chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt,
human_message_prompt])

# get a chat completion from the formatted messages
chat(chat_prompt.format_prompt(input_language="English",
output_language="French", text="I love programming.").to_messages())
# -> AIMessage(content="J'aime programmer.", additional_kwargs={})
```

Chains with Chat Models

The `LLMChain` discussed in the above section can be used with chat models as well:

```
from langchain.chat_models import ChatOpenAI
from langchain import LLMChain
from langchain.prompts.chat import (
```

```
    HumanMessagePromptTemplate,
)

chat = ChatOpenAI(temperature=0)

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)
chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt,
human_message_prompt])

chain = LLMChain(llm=chat, prompt=chat_prompt)
chain.run(input_language="English", output_language="French", text="I love
programming.")
# -> "J'aime programmer."
```

Agents with Chat Models

Agents can also be used with chat models, you can initialize one using `AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION` as the agent type.

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.chat_models import ChatOpenAI
from langchain.llms import OpenAI

# First, let's load the language model we're going to use to control the
agent.
chat = ChatOpenAI(temperature=0)

# Next, let's load some tools to use. Note that the `llm-math` tool uses an
LLM, so we need to pass that in.
llm = OpenAI(temperature=0)
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, chat,
agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

# Now let's test it out!
agent.run("Who is Olivia Wilde's boyfriend? What is his current age r
to the 0.23 power?")
```

⌘ + K

Skip to main content

```
> Entering new AgentExecutor chain...
Thought: I need to use a search engine to find Olivia Wilde's boyfriend and
a calculator to raise his age to the 0.23 power.
Action:
{
  "action": "Search",
  "action_input": "Olivia Wilde boyfriend"
}

Observation: Sudeikis and Wilde's relationship ended in November 2020.
Wilde was publicly served with court documents regarding child custody
while she was presenting Don't Worry Darling at CinemaCon 2022. In January
2021, Wilde began dating singer Harry Styles after meeting during the
filming of Don't Worry Darling.
Thought:I need to use a search engine to find Harry Styles' current age.
Action:
{
  "action": "Search",
  "action_input": "Harry Styles age"
}

Observation: 29 years
Thought:Now I need to calculate 29 raised to the 0.23 power.
Action:
{
  "action": "Calculator",
  "action_input": "29^0.23"
}

Observation: Answer: 2.169459462491557

Thought:I now know the final answer.
Final Answer: 2.169459462491557

> Finished chain.
'2.169459462491557'
```

Memory: Add State to Chains and Agents

You can use Memory with chains and agents initialized with chat models. The main difference between this and Memory for LLMs is that rather than trying to condense all previous messages into a string, we can keep them as their own unique memory object.

```
from langchain.prompts import (
    ChatPromptTemplate,
    MessagesPlaceholder,
```

```
)
from langchain.chains import ConversationChain
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory

prompt = ChatPromptTemplate.from_messages([
    SystemMessagePromptTemplate.from_template("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."),
    MessagesPlaceholder(variable_name="history"),
    HumanMessagePromptTemplate.from_template("{input}")
])

llm = ChatOpenAI(temperature=0)
memory = ConversationBufferMemory(return_messages=True)
conversation = ConversationChain(memory=memory, prompt=prompt, llm=llm)

conversation.predict(input="Hi there!")
# -> 'Hello! How can I assist you today?'


conversation.predict(input="I'm doing well! Just having a conversation with
an AI.")
# -> "That sounds like fun! I'm happy to chat with you. Is there anything
specific you'd like to talk about?"

conversation.predict(input="Tell me about yourself.")
# -> "Sure! I am an AI language model created by OpenAI. I was trained on a
large dataset of text from the internet, which allows me to understand and
generate human-like language. I can answer questions, provide information,
and even have conversations like this one. Is there anything else you'd
like to know about me?"
```