Quickstart

Installation

To install LangChain run:

Pip Conda

```
pip install langchain
```

For more details, see our Installation guide.

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. 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

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

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

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

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

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

```python
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

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.
- 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. For a list of supported agents and their specifications, see here.

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

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?"
```