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

Here's how to install LangChain, set up your environment, and start building.

We recommend following our Quickstart guide to familiarize yourself with the framework by building your first LangChain application.

***Note**: These docs are for the LangChain Python package. For documentation on LangChain.js, the JS/TS version, head here.*

Modules

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

Model I/O

Interface with language models

Data connection

Interface with application-specific data

Chains

Construct sequences of calls

Agents

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

Memory

Persist application state between runs of a chain

Callbacks

Log and stream intermediate steps of any chain

Examples, ecosystem, and resources

Use cases

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

- Chatbots
- Answering questions using sources
- Analyzing structured data
- and much more...

Guides

Learn best practices for developing with LangChain.

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 and dependent repos.

Additional resources

Our community is full of prolific developers, creative builders, and fantastic teachers. Check out YouTube tutorials for great tutorials from folks in the community, and Gallery for a list of awesome LangChain projects, compiled by the folks at KyroLabs.

Support

Join us on GitHub or Discord to ask questions, share feedback, meet other developers building with LangChain, and dream about the future of LLM's.

API reference

Head to the reference section for full documentation of all classes and methods in the LangChain Python package.

Caching

LangChain provides an optional caching layer for Chat Models. This is useful for two reasons:

It can save you money by reducing the number of API calls you make to the LLM provider, if you're often requesting the same completion multiple times. It can speed up your application by reducing the number of API calls you make to the LLM provider.

```
import langchain
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI()
```

In Memory Cache

```
from langchain.cache import InMemoryCache
langchain.llm_cache = InMemoryCache()

# The first time, it is not yet in cache, so it should take longer
llm.predict("Tell me a joke")
```

```
    CPU times: user 35.9 ms, sys: 28.6 ms, total: 64.6 ms
    Wall time: 4.83 s


    "\n\nWhy couldn't the bicycle stand up by itself? It was...two
tired!"
```

```
# The second time it is, so it goes faster
llm.predict("Tell me a joke")
```

```
    CPU times: user 238 µs, sys: 143 µs, total: 381 µs
    Wall time: 1.76 ms
```

```
    '\n\nWhy did the chicken cross the road?\n\nTo get to the other
side.'
```

SQLite Cache

```
rm .langchain.db
```

```
# We can do the same thing with a SQLite cache
from langchain.cache import SQLiteCache
langchain.llm_cache = SQLiteCache(database_path=".langchain.db")
```

```
# The first time, it is not yet in cache, so it should take longer
llm.predict("Tell me a joke")
```

```
    CPU times: user 17 ms, sys: 9.76 ms, total: 26.7 ms
    Wall time: 825 ms


    '\n\nWhy did the chicken cross the road?\n\nTo get to the other
side.'
```

```
# The second time it is, so it goes faster
llm.predict("Tell me a joke")
```

```
    CPU times: user 2.46 ms, sys: 1.23 ms, total: 3.7 ms
    Wall time: 2.67 ms


    '\n\nWhy did the chicken cross the road?\n\nTo get to the other
side.'
```

Human input Chat Model

Along with HumanInputLLM, LangChain also provides a pseudo Chat Model class that can be used for testing, debugging, or educational purposes. This allows you to mock out calls to the Chat Model and simulate how a human would respond if they received the messages.

In this notebook, we go over how to use this.

We start this with using the HumanInputChatModel in an agent.

```
from langchain.chat_models.human import HumanInputChatModel
```

API Reference:

- HumanInputChatModel from `langchain.chat_models.human`

Since we will use the `WikipediaQueryRun` tool in this notebook, you might need to install the `wikipedia` package if you haven't done so already.

```
%pip install wikipedia
```

```
/Users/mskim58/dev/research/chatbot/github/langchain/.venv/bin/python:
No module named pip
    Note: you may need to restart the kernel to use updated packages.
```

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

API Reference:

- load_tools from `langchain.agents`
- initialize_agent from `langchain.agents`
- AgentType from `langchain.agents`

```
tools = load_tools(["wikipedia"])
llm = HumanInputChatModel()
```

```
agent = initialize_agent(
    tools, llm, agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION,
verbose=True
)
```

```
agent("What is Bocchi the Rock?")
```

```
    > Entering new  chain...

     ======= start of message =======


    type: system
    data:
      content: "Answer the following questions as best you can. You
have access to the following tools:\n\nWikipedia: A wrapper around
Wikipedia. Useful for when you need to answer general questions about
people, places, companies, facts, historical events, or other subjects.
Input should be a search query.\n\nThe way you use the tools is by
specifying a json blob.\nSpecifically, this json should have a `action`
key (with the name of the tool to use) and a `action_input` key (with
the input to the tool going here).\n\nThe only values that should be in
the \"action\" field are: Wikipedia\n\nThe $JSON_BLOB should only
contain a SINGLE action, do NOT return a list of multiple actions. Here
is an example of a valid $JSON_BLOB:\n\n```\n{\n  \"action\":
$TOOL_NAME,\n  \"action_input\": $INPUT\n}\n```\n\nALWAYS use the
following format:\n\nQuestion: the input question you must
answer\nThought: you should always think about what to
do\nAction:\n```\n$JSON_BLOB\n```\nObservation: the result of the
action\n... (this Thought/Action/Observation can repeat N
times)\nThought: I now know the final answer\nFinal Answer: the final
answer to the original input question\n\nBegin! Reminder to always use
the exact characters `Final Answer` when responding."
      additional_kwargs: {}

    ======= end of message =======
```

```
 ======= start of message =======


type: human
data:
  content: 'What is Bocchi the Rock?


    '
  additional_kwargs: {}
  example: false

======= end of message =======


Action:
```
{
  "action": "Wikipedia",
  "action_input": "What is Bocchi the Rock?"
}
```
Observation: Page: Bocchi the Rock!
Summary: Bocchi the Rock! (ぼっち・ざ・ろっく!, Botchi Za Rokku!) is a Japanese four-panel manga series written and illustrated by Aki Hamaji. It has been serialized in Houbunsha's seinen manga magazine Manga Time Kirara Max since December 2017. Its chapters have been collected in five tankōbon volumes as of November 2022.
An anime television series adaptation produced by CloverWorks aired from October to December 2022. The series has been praised for its writing, comedy, characters, and depiction of social anxiety, with the anime's visual creativity receiving acclaim.

Page: Hitori Bocchi no Marumaru Seikatsu
Summary: Hitori Bocchi no Marumaru Seikatsu (Japanese: ひとりぼっちの○○生活, lit. "Bocchi Hitori's ____ Life" or "The ____ Life of Being Alone") is a Japanese yonkoma manga series written and illustrated by Katsuwo. It was serialized in ASCII Media Works' Comic Dengeki Daioh "g" magazine from September 2013 to April 2021. Eight tankōbon volumes have been released. An anime television series adaptation by C2C aired from April to June 2019.

Page: Kessoku Band (album)
Summary: Kessoku Band (Japanese: 結束バンド, Hepburn: Kessoku Bando) is the debut studio album by Kessoku Band, a fictional musical group
```

from the anime television series Bocchi the Rock!, released digitally on December 25, 2022, and physically on CD on December 28 by Aniplex. Featuring vocals from voice actresses Yoshino Aoyama, Sayumi Suzushiro, Saku Mizuno, and Ikumi Hasegawa, the album consists of 14 tracks previously heard in the anime, including a cover of Asian Kung-Fu Generation's "Rockn' Roll, Morning Light Falls on You", as well as newly recorded songs; nine singles preceded the album's physical release. Commercially, Kessoku Band peaked at number one on the Billboard Japan Hot Albums Chart and Oricon Albums Chart, and was certified gold by the Recording Industry Association of Japan.

Thought:
======= start of message =======

type: system
data:
content: "Answer the following questions as best you can. You have access to the following tools:\n\nWikipedia: A wrapper around Wikipedia. Useful for when you need to answer general questions about people, places, companies, facts, historical events, or other subjects. Input should be a search query.\n\nThe way you use the tools is by specifying a json blob.\nSpecifically, this json should have a `action` key (with the name of the tool to use) and a `action_input` key (with the input to the tool going here).\n\nThe only values that should be in the \"action\" field are: Wikipedia\n\nThe $JSON_BLOB should only contain a SINGLE action, do NOT return a list of multiple actions. Here is an example of a valid $JSON_BLOB:\n\n```\n{\n \"action\": $TOOL_NAME,\n \"action_input\": $INPUT\n}\n```\n\nALWAYS use the following format:\n\nQuestion: the input question you must answer\nThought: you should always think about what to do\nAction:\n```\n$JSON_BLOB\n```\nObservation: the result of the action\n... (this Thought/Action/Observation can repeat N times)\nThought: I now know the final answer\nFinal Answer: the final answer to the original input question\n\nBegin! Reminder to always use the exact characters `Final Answer` when responding."
additional_kwargs: {}

======= end of message =======

======= start of message =======

type: human

```
    data:
      content: "What is Bocchi the Rock?\n\nThis was your previous work
(but I haven't seen any of it! I only see what you return as final
answer):\nAction:\n```\n{\n  \"action\": \"Wikipedia\",\n
\"action_input\": \"What is Bocchi the Rock?\"\n}\n```\nObservation:
Page: Bocchi the Rock!\nSummary: Bocchi the Rock! (ぼっち・ざ・ろっく!,
Botchi Za Rokku!) is a Japanese four-panel manga series written and
illustrated by Aki Hamaji. It has been serialized in Houbunsha's seinen
manga magazine Manga Time Kirara Max since December 2017. Its chapters
have been collected in five tankōbon volumes as of November 2022.\nAn
anime television series adaptation produced by CloverWorks aired from
October to December 2022. The series has been praised for its writing,
comedy, characters, and depiction of social anxiety, with the anime's
visual creativity receiving acclaim.\n\nPage: Hitori Bocchi no Marumaru
Seikatsu\nSummary: Hitori Bocchi no Marumaru Seikatsu (Japanese: ひとりぼ
っちの○○生活, lit. \"Bocchi Hitori's ____ Life\" or \"The ____ Life of
Being Alone\") is a Japanese yonkoma manga series written and
illustrated by Katsuwo. It was serialized in ASCII Media Works' Comic
Dengeki Daioh \"g\" magazine from September 2013 to April 2021. Eight
tankōbon volumes have been released. An anime television series
adaptation by C2C aired from April to June 2019.\n\nPage: Kessoku Band
(album)\nSummary: Kessoku Band (Japanese: 結束バンド, Hepburn: Kessoku
Bando) is the debut studio album by Kessoku Band, a fictional musical
group from the anime television series Bocchi the Rock!, released
digitally on December 25, 2022, and physically on CD on December 28 by
Aniplex. Featuring vocals from voice actresses Yoshino Aoyama, Sayumi
Suzushiro, Saku Mizuno, and Ikumi Hasegawa, the album consists of 14
tracks previously heard in the anime, including a cover of Asian Kung-
Fu Generation's \"Rockn' Roll, Morning Light Falls on You\", as well as
newly recorded songs; nine singles preceded the album's physical
release. Commercially, Kessoku Band peaked at number one on the
Billboard Japan Hot Albums Chart and Oricon Albums Chart, and was
certified gold by the Recording Industry Association of
Japan.\n\n\nThought:"
      additional_kwargs: {}
      example: false

    ======= end of message =======


    This finally works.
    Final Answer: Bocchi the Rock! is a four-panel manga series and
anime television series. The series has been praised for its writing,
comedy, characters, and depiction of social anxiety, with the anime's
visual creativity receiving acclaim.

    > Finished chain.
```

```
    {'input': 'What is Bocchi the Rock?',
     'output': "Bocchi the Rock! is a four-panel manga series and anime
television series. The series has been praised for its writing, comedy,
characters, and depiction of social anxiety, with the anime's visual
creativity receiving acclaim."}
```

Prompts

Prompts for Chat models are built around messages, instead of just plain text.

You can make use of templating by using a `MessagePromptTemplate`. 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.

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 import PromptTemplate
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    AIMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

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'adore la programmation.", additional_kwargs=
{})
```

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 = SystemMessagePromptTemplate(prompt=prompt)
```

Output parsers

Language models output text. But many times you may want to get more structured information than just text back. This is where output parsers come in.

Output parsers are classes that help structure language model responses. There are two main methods an output parser must implement:

- "Get format instructions": A method which returns a string containing instructions for how the output of a language model should be formatted.
- "Parse": A method which takes in a string (assumed to be the response from a language model) and parses it into some structure.

And then one optional one:

- "Parse with prompt": A method which takes in a string (assumed to be the response from a language model) and a prompt (assumed to the prompt that generated such a response) and parses it into some structure. The prompt is largely provided in the event the OutputParser wants to retry or fix the output in some way, and needs information from the prompt to do so.

Get started

Below we go over the main type of output parser, the `PydanticOutputParser`.

```
from langchain.prompts import PromptTemplate, ChatPromptTemplate,
HumanMessagePromptTemplate
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI

from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field, validator
from typing import List
```

```
model_name = 'text-davinci-003'
temperature = 0.0
```

```
model = OpenAI(model_name=model_name, temperature=temperature)
```

```
# Define your desired data structure.
class Joke(BaseModel):
    setup: str = Field(description="question to set up a joke")
    punchline: str = Field(description="answer to resolve the joke")

    # You can add custom validation logic easily with Pydantic.
    @validator('setup')
    def question_ends_with_question_mark(cls, field):
        if field[-1] != '?':
            raise ValueError("Badly formed question!")
        return field
```

```
# Set up a parser + inject instructions into the prompt template.
parser = PydanticOutputParser(pydantic_object=Joke)
```

```
prompt = PromptTemplate(
    template="Answer the user
query.\n{format_instructions}\n{query}\n",
    input_variables=["query"],
    partial_variables={"format_instructions":
parser.get_format_instructions()}
)
```

```
# And a query intended to prompt a language model to populate the data
structure.
joke_query = "Tell me a joke."
_input = prompt.format_prompt(query=joke_query)
```

```
output = model(_input.to_string())
```

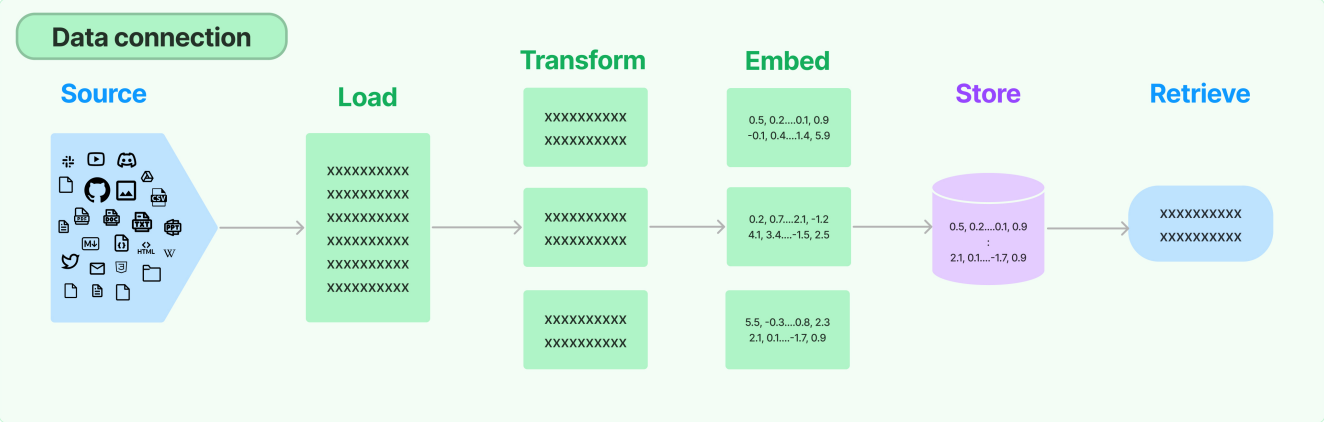
```
parser.parse(output)
```

```
    Joke(setup='Why did the chicken cross the road?', punchline='To get
to the other side!')
```

Data connection

Many LLM applications require user-specific data that is not part of the model's training set. LangChain gives you the building blocks to load, transform, store and query your data via:

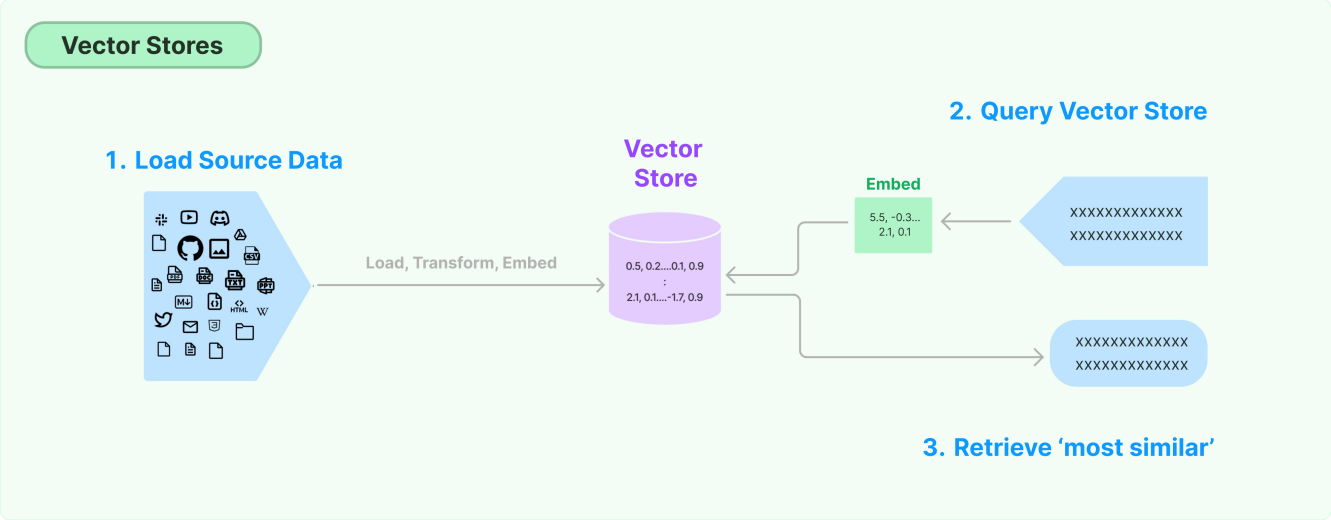
- Document loaders: Load documents from many different sources
- Document transformers: Split documents, convert documents into Q&A format, drop redundant documents, and more
- Text embedding models: Take unstructured text and turn it into a list of floating point numbers
- Vector stores: Store and search over embedded data
- Retrievers: Query your data



Vector stores

> **INFO**
>
> Head to Integrations for documentation on built-in integrations with 3rd-party vector stores.

One of the most common ways to store and search over unstructured data is to embed it and store the resulting embedding vectors, and then at query time to embed the unstructured query and retrieve the embedding vectors that are 'most similar' to the embedded query. A vector store takes care of storing embedded data and performing vector search for you.



Get started

This walkthrough showcases basic functionality related to VectorStores. A key part of working with vector stores is creating the vector to put in them, which is usually created via embeddings. Therefore, it is recommended that you familiarize yourself with the text embedding model interfaces before diving into this.

There are many great vector store options, here are a few that are free, open-source, and run entirely on your local machine. Review all integrations for many great hosted offerings.

Chroma FAISS Lance

This walkthrough uses the `chroma` vector database, which runs on your local machine as a library.

```
pip install chromadb
```

We want to use OpenAIEmbeddings so we have to get the OpenAI API Key.

```
import os
import getpass

os.environ['OPENAI_API_KEY'] = getpass.getpass('OpenAI API Key:')
```

```
from langchain.document_loaders import TextLoader
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import Chroma

# Load the document, split it into chunks, embed each chunk and load it
into the vector store.
raw_documents = TextLoader('../../../state_of_the_union.txt').load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
documents = text_splitter.split_documents(raw_documents)
db = Chroma.from_documents(documents, OpenAIEmbeddings())
```

Similarity search

```
query = "What did the president say about Ketanji Brown Jackson"
docs = db.similarity_search(query)
print(docs[0].page_content)
```

```
    Tonight. I call on the Senate to: Pass the Freedom to Vote Act.
Pass the John Lewis Voting Rights Act. And while you're at it, pass the
Disclose Act so Americans can know who is funding our elections.

    Tonight, I'd like to honor someone who has dedicated his life to
serve this country: Justice Stephen Breyer—an Army veteran,
Constitutional scholar, and retiring Justice of the United States
Supreme Court. Justice Breyer, thank you for your service.

    One of the most serious constitutional responsibilities a President
```

```
has is nominating someone to serve on the United States Supreme Court.

    And I did that 4 days ago, when I nominated Circuit Court of
Appeals Judge Ketanji Brown Jackson. One of our nation's top legal
minds, who will continue Justice Breyer's legacy of excellence.
```

Similarity search by vector

It is also possible to do a search for documents similar to a given embedding vector using `similarity_search_by_vector` which accepts an embedding vector as a parameter instead of a string.

```
embedding_vector = OpenAIEmbeddings().embed_query(query)
docs = db.similarity_search_by_vector(embedding_vector)
print(docs[0].page_content)
```

The query is the same, and so the result is also the same.

```
    Tonight. I call on the Senate to: Pass the Freedom to Vote Act.
Pass the John Lewis Voting Rights Act. And while you're at it, pass the
Disclose Act so Americans can know who is funding our elections.

    Tonight, I'd like to honor someone who has dedicated his life to
serve this country: Justice Stephen Breyer—an Army veteran,
Constitutional scholar, and retiring Justice of the United States
Supreme Court. Justice Breyer, thank you for your service.

    One of the most serious constitutional responsibilities a President
has is nominating someone to serve on the United States Supreme Court.

    And I did that 4 days ago, when I nominated Circuit Court of
Appeals Judge Ketanji Brown Jackson. One of our nation's top legal
minds, who will continue Justice Breyer's legacy of excellence.
```

Asynchronous operations

Vector stores are usually run as a separate service that requires some IO operations, and therefore they might be called asynchronously. That gives performance benefits as you don't waste time waiting for responses from external services. That might also be important if you work with an asynchronous framework, such as FastAPI.

Langchain supports async operation on vector stores. All the methods might be called using their async counterparts, with the prefix `a`, meaning `async`.

`Qdrant` is a vector store, which supports all the async operations, thus it will be used in this walkthrough.

```
pip install qdrant-client
```

```
from langchain.vectorstores import Qdrant
```

Create a vector store asynchronously

```
db = await Qdrant.afrom_documents(documents, embeddings,
"http://localhost:6333")
```

Similarity search

```
query = "What did the president say about Ketanji Brown Jackson"
docs = await db.asimilarity_search(query)
print(docs[0].page_content)
```

```
    Tonight. I call on the Senate to: Pass the Freedom to Vote Act.
Pass the John Lewis Voting Rights Act. And while you're at it, pass the
Disclose Act so Americans can know who is funding our elections.

    Tonight, I'd like to honor someone who has dedicated his life to
serve this country: Justice Stephen Breyer—an Army veteran,
Constitutional scholar, and retiring Justice of the United States
Supreme Court. Justice Breyer, thank you for your service.

    One of the most serious constitutional responsibilities a President
has is nominating someone to serve on the United States Supreme Court.

    And I did that 4 days ago, when I nominated Circuit Court of
Appeals Judge Ketanji Brown Jackson. One of our nation's top legal
minds, who will continue Justice Breyer's legacy of excellence.
```

Similarity search by vector

```
embedding_vector = embeddings.embed_query(query)
docs = await db.asimilarity_search_by_vector(embedding_vector)
```

Maximum marginal relevance search (MMR)

Maximal marginal relevance optimizes for similarity to query AND diversity among selected documents. It is also supported in async API.

```
query = "What did the president say about Ketanji Brown Jackson"
found_docs = await qdrant.amax_marginal_relevance_search(query, k=2,
fetch_k=10)
for i, doc in enumerate(found_docs):
    print(f"{i + 1}.", doc.page_content, "\n")
```

```
1. Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass
the John Lewis Voting Rights Act. And while you're at it, pass the
Disclose Act so Americans can know who is funding our elections.

Tonight, I'd like to honor someone who has dedicated his life to serve
this country: Justice Stephen Breyer—an Army veteran, Constitutional
scholar, and retiring Justice of the United States Supreme Court.
Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has
is nominating someone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals
Judge Ketanji Brown Jackson. One of our nation's top legal minds, who
will continue Justice Breyer's legacy of excellence.

2. We can't change how divided we've been. But we can change how we
move forward—on COVID-19 and other issues we must face together.

I recently visited the New York City Police Department days after the
funerals of Officer Wilbert Mora and his partner, Officer Jason Rivera.

They were responding to a 9-1-1 call when a man shot and killed them
with a stolen gun.

Officer Mora was 27 years old.

Officer Rivera was 22.
```

Both Dominican Americans who’d grown up on the same streets they later chose to patrol as police officers.

I spoke with their families and told them that we are forever in debt for their sacrifice, and we will carry on their mission to restore the trust and safety every community deserves.

I’ve worked on these issues a long time.

I know what works: Investing in crime preventionand community police officers who’ll walk the beat, who’ll know the neighborhood, and who can restore trust and safety.

Chains

Using an LLM in isolation is fine for simple applications, but more complex applications require chaining LLMs - either with each other or with other components.

LangChain provides the **Chain** interface for such "chained" applications. We define a Chain very generically as a sequence of calls to components, which can include other chains. The base interface is simple:

```
class Chain(BaseModel, ABC):
    """Base interface that all chains should implement."""

    memory: BaseMemory
    callbacks: Callbacks

    def __call__(
        self,
        inputs: Any,
        return_only_outputs: bool = False,
        callbacks: Callbacks = None,
    ) -> Dict[str, Any]:
        ...
```

This idea of composing components together in a chain is simple but powerful. It drastically simplifies and makes more modular the implementation of complex applications, which in turn makes it much easier to debug, maintain, and improve your applications.

For more specifics check out:

- How-to for walkthroughs of different chain features
- Foundational to get acquainted with core building block chains
- Document to learn how to incorporate documents into chains
- Popular chains for the most common use cases
- Additional to see some of the more advanced chains and integrations that you can use out of the box

Why do we need chains?

Chains allow us to combine multiple components together to create a single, coherent application. For example, we can create a chain that takes user input, formats it with a PromptTemplate, and then passes the formatted response to an LLM. We can build more complex chains by combining multiple chains together, or by combining chains with other components.

Get started

Using `LLMChain`

The `LLMChain` is most basic building block chain. It takes in a prompt template, formats it with the user input and returns the response from an LLM.

To use the `LLMChain`, first create a prompt template.

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

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)

# Run the chain only specifying the input variable.
print(chain.run("colorful socks"))
```

```
    Colorful Toes Co.
```

If there are multiple variables, you can input them all at once using a dictionary.

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

```
)
chain = LLMChain(llm=llm, prompt=prompt)
print(chain.run({
    'company': "ABC Startup",
    'product': "colorful socks"
    }))
```

```
    Socktopia Colourful Creations.
```

You can use a chat model in an `LLMChain` as well:

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts.chat import (
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
)
human_message_prompt = HumanMessagePromptTemplate(
        prompt=PromptTemplate(
            template="What is a good name for a company that makes
{product}?",
            input_variables=["product"],
        )
    )
chat_prompt_template =
ChatPromptTemplate.from_messages([human_message_prompt])
chat = ChatOpenAI(temperature=0.9)
chain = LLMChain(llm=chat, prompt=chat_prompt_template)
print(chain.run("colorful socks"))
```

```
    Rainbow Socks Co.
```

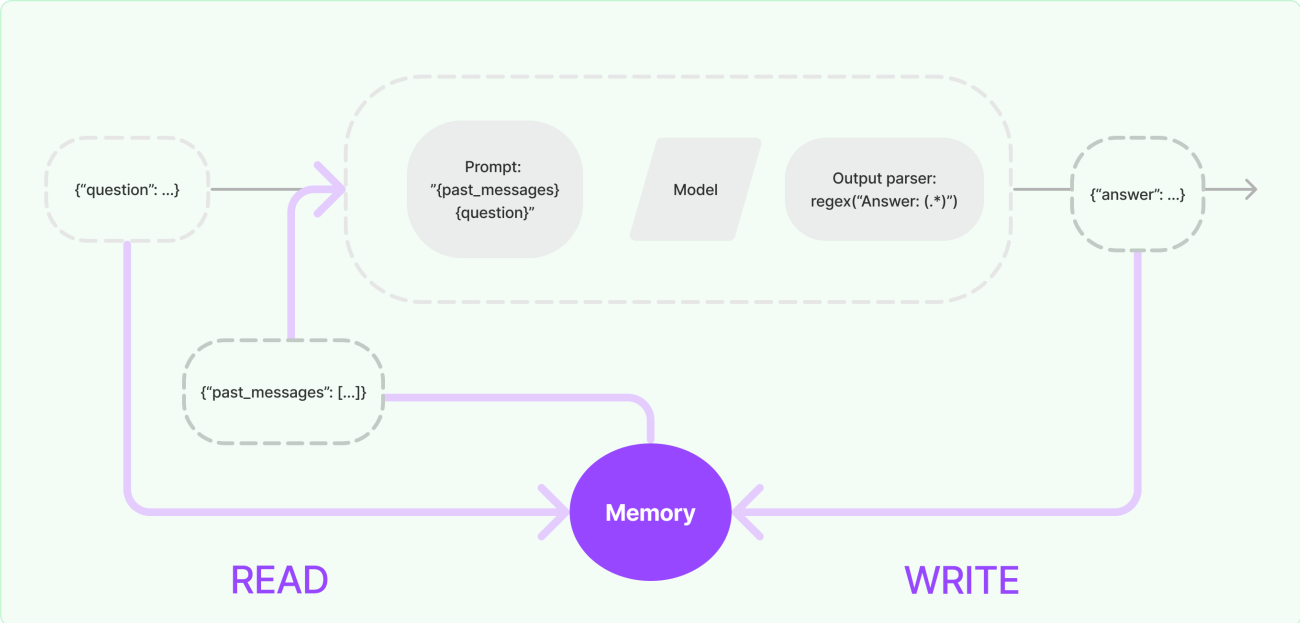
Memory

Most LLM applications have a conversational interface. An essential component of a conversation is being able to refer to information introduced earlier in the conversation. At bare minimum, a conversational system should be able to access some window of past messages directly. A more complex system will need to have a world model that it is constantly updating, which allows it to do things like maintain information about entities and their relationships.

We call this ability to store information about past interactions "memory". LangChain provides a lot of utilities for adding memory to a system. These utilities can be used by themselves or incorporated seamlessly into a chain.

A memory system needs to support two basic actions: reading and writing. Recall that every chain defines some core execution logic that expects certain inputs. Some of these inputs come directly from the user, but some of these inputs can come from memory. A chain will interact with its memory system twice in a given run.

1. AFTER receiving the initial user inputs but BEFORE executing the core logic, a chain will READ from its memory system and augment the user inputs.
2. AFTER executing the core logic but BEFORE returning the answer, a chain will WRITE the inputs and outputs of the current run to memory, so that they can be referred to in future runs.



Building memory into a system

The two core design decisions in any memory system are:

- How state is stored
- How state is queried

Storing: List of chat messages

Underlying any memory is a history of all chat interactions. Even if these are not all used directly, they need to be stored in some form. One of the key parts of the LangChain memory module is a series of integrations for storing these chat messages, from in-memory lists to persistent databases.

- Chat message storage: How to work with Chat Messages, and the various integrations offered

Querying: Data structures and algorithms on top of chat messages

Keeping a list of chat messages is fairly straight-forward. What is less straight-forward are the data structures and algorithms built on top of chat messages that serve a view of those messages that is most useful.

A very simply memory system might just return the most recent messages each run. A slightly more complex memory system might return a succinct summary of the past K messages. An even more sophisticated system might extract entities from stored messages and only return information about entities referenced in the current run.

Each application can have different requirements for how memory is queried. The memory module should make it easy to both get started with simple memory systems and write your own custom systems if needed.

- Memory types: The various data structures and algorithms that make up the memory types LangChain supports

Get started

Let's take a look at what Memory actually looks like in LangChain. Here we'll cover the basics of interacting with an arbitrary memory class.

Let's take a look at how to use ConversationBufferMemory in chains. ConversationBufferMemory is an extremely simple form of memory that just keeps a list of chat messages in a buffer and passes those into the prompt template.

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory()
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

When using memory in a chain, there are a few key concepts to understand. Note that here we cover general concepts that are useful for most types of memory. Each individual memory type may very well have its own parameters and concepts that are necessary to understand.

What variables get returned from memory

Before going into the chain, various variables are read from memory. This have specific names which need to align with the variables the chain expects. You can see what these variables are by calling `memory.load_memory_variables({})`. Note that the empty dictionary that we pass in is just a placeholder for real variables. If the memory type you are using is dependent upon the input variables, you may need to pass some in.

```
memory.load_memory_variables({})
```

```
    {'history': "Human: hi!\nAI: whats up?"}
```

In this case, you can see that `load_memory_variables` returns a single key, `history`. This means that your chain (and likely your prompt) should expect and input named `history`. You can usually control this variable through parameters on the memory class. For example, if you want the memory variables to be returned in the key `chat_history` you can do:

```
memory = ConversationBufferMemory(memory_key="chat_history")
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

```
    {'chat_history': "Human: hi!\nAI: whats up?"}
```

The parameter name to control these keys may vary per memory type, but it's important to understand that (1) this is controllable, (2) how to control it.

Whether memory is a string or a list of messages

One of the most common types of memory involves returning a list of chat messages. These can either be returned as a single string, all concatenated together (useful when they will be passed in LLMs) or a list of ChatMessages (useful when passed into ChatModels).

By default, they are returned as a single string. In order to return as a list of messages, you can set `return_messages=True`

```
memory = ConversationBufferMemory(return_messages=True)
memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")
```

```
    {'history': [HumanMessage(content='hi!', additional_kwargs={},
example=False),
  AIMessage(content='whats up?', additional_kwargs={}, example=False)]}
```

What keys are saved to memory

Often times chains take in or return multiple input/output keys. In these cases, how can we know which keys we want to save to the chat message history? This is generally controllable by `input_key` and `output_key` parameters on the memory types. These default to None - and if there is only one input/output key it is known to just use that. However, if there are multiple input/output keys then you MUST specify the name of which one to use

End to end example

Finally, let's take a look at using this in a chain. We'll use an LLMChain, and show working with both an LLM and a ChatModel.

Using an LLM

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory


llm = OpenAI(temperature=0)
```

```
# Notice that "chat_history" is present in the prompt template
template = """You are a nice chatbot having a conversation with a
human.

Previous conversation:
{chat_history}

New human question: {question}
Response:"""
prompt = PromptTemplate.from_template(template)
# Notice that we need to align the `memory_key`
memory = ConversationBufferMemory(memory_key="chat_history")
conversation = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory
)
```

```
# Notice that we just pass in the `question` variables - `chat_history`
gets populated by memory
conversation({"question": "hi"})
```

Using a ChatModel

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import (
    ChatPromptTemplate,
    MessagesPlaceholder,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory


llm = ChatOpenAI()
prompt = ChatPromptTemplate(
    messages=[
        SystemMessagePromptTemplate.from_template(
            "You are a nice chatbot having a conversation with a
human."
        ),
        # The `variable_name` here is what must align with memory
        MessagesPlaceholder(variable_name="chat_history"),
```

```
        HumanMessagePromptTemplate.from_template("{question}")
    ]
)
# Notice that we `return_messages=True` to fit into the
MessagesPlaceholder
# Notice that `"chat_history"` aligns with the MessagesPlaceholder
name.
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True)
conversation = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory
)
```

```
# Notice that we just pass in the `question` variables - `chat_history`
gets populated by memory
conversation({"question": "hi"})
```

Next steps

And that's it for getting started! Please see the other sections for walkthroughs of more advanced topics, like custom memory, multiple memories, and more.

Chat Messages

> **INFO**
>
> Head to Integrations for documentation on built-in memory integrations with 3rd-party databases and tools.

One of the core utility classes underpinning most (if not all) memory modules is the `ChatMessageHistory` class. This is a super lightweight wrapper which exposes convenience methods for saving Human messages, AI messages, and then fetching them all.

You may want to use this class directly if you are managing memory outside of a chain.

```
from langchain.memory import ChatMessageHistory

history = ChatMessageHistory()

history.add_user_message("hi!")

history.add_ai_message("whats up?")
```

```
history.messages
```

```
    [HumanMessage(content='hi!', additional_kwargs={}),
     AIMessage(content='whats up?', additional_kwargs={})]
```

How to add Memory to an LLMChain

This notebook goes over how to use the Memory class with an LLMChain. For the purposes of this walkthrough, we will add the ConversationBufferMemory class, although this can be any memory class.

```
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.memory import ConversationBufferMemory
from langchain.prompts import PromptTemplate
```

API Reference:

- LLMChain from `langchain.chains`
- OpenAI from `langchain.llms`
- ConversationBufferMemory from `langchain.memory`
- PromptTemplate from `langchain.prompts`

The most important step is setting up the prompt correctly. In the below prompt, we have two input keys: one for the actual input, another for the input from the Memory class. Importantly, we make sure the keys in the PromptTemplate and the ConversationBufferMemory match up (`chat_history`).

```
template = """You are a chatbot having a conversation with a human.

{chat_history}
Human: {human_input}
Chatbot:"""

prompt = PromptTemplate(
    input_variables=["chat_history", "human_input"], template=template
)
memory = ConversationBufferMemory(memory_key="chat_history")
```

```
llm = OpenAI()
llm_chain = LLMChain(
    llm=llm,
    prompt=prompt,
```

```
    verbose=True,
    memory=memory,
)
```

```
llm_chain.predict(human_input="Hi there my friend")
```

```
> Entering new LLMChain chain...
Prompt after formatting:
You are a chatbot having a conversation with a human.

Human: Hi there my friend
Chatbot:

> Finished chain.
```

```
' Hi there! How can I help you today?'
```

```
llm_chain.predict(human_input="Not too bad - how are you?")
```

```
> Entering new LLMChain chain...
Prompt after formatting:
You are a chatbot having a conversation with a human.

Human: Hi there my friend
AI:  Hi there! How can I help you today?
Human: Not too bad - how are you?
Chatbot:

> Finished chain.
```

```
    " I'm doing great, thanks for asking! How are you doing?"
```

Adding Memory to a Chat Model-based LLMChain

The above works for completion-style `LLM`s, but if you are using a chat model, you will likely get better performance using structured chat messages. Below is an example.

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import SystemMessage
from langchain.prompts import ChatPromptTemplate,
HumanMessagePromptTemplate, MessagesPlaceholder
```

API Reference:

- ChatOpenAI from `langchain.chat_models`
- SystemMessage from `langchain.schema`
- ChatPromptTemplate from `langchain.prompts`
- HumanMessagePromptTemplate from `langchain.prompts`
- MessagesPlaceholder from `langchain.prompts`

We will use the ChatPromptTemplate class to set up the chat prompt.

The from_messages method creates a ChatPromptTemplate from a list of messages (e.g., SystemMessage, HumanMessage, AIMessage, ChatMessage, etc.) or message templates, such as the MessagesPlaceholder below.

The configuration below makes it so the memory will be injected to the middle of the chat prompt, in the "chat_history" key, and the user's inputs will be added in a human/user message to the end of the chat prompt.

```
prompt = ChatPromptTemplate.from_messages([
    SystemMessage(content="You are a chatbot having a conversation with
a human."), # The persistent system prompt
    MessagesPlaceholder(variable_name="chat_history"), # Where the
memory will be stored.
    HumanMessagePromptTemplate.from_template("{human_input}"), # Where
the human input will injectd
])
```

```
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True)
```

```
llm = ChatOpenAI()

chat_llm_chain = LLMChain(
    llm=llm,
    prompt=prompt,
    verbose=True,
    memory=memory,
)
```

```
chat_llm_chain.predict(human_input="Hi there my friend")
```

```
    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a chatbot having a conversation with a human.
    Human: Hi there my friend

    > Finished chain.
```

```
    'Hello! How can I assist you today, my friend?'
```

```
chat_llm_chain.predict(human_input="Not too bad - how are you?")
```

```
    > Entering new LLMChain chain...
    Prompt after formatting:
    System: You are a chatbot having a conversation with a human.
    Human: Hi there my friend
    AI: Hello! How can I assist you today, my friend?
    Human: Not too bad - how are you?
```

```
    > Finished chain.
```

```
    "I'm an AI chatbot, so I don't have feelings, but I'm here to help
and chat with you! Is there something specific you would like to talk
about or any questions I can assist you with?"
```

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.

The core building block of LangChain applications is the LLMChain. This combines three things:

- LLM: The language model is the core reasoning engine here. In order to work with LangChain, you need to understand the different types of language models and how to work with them.
- Prompt Templates: This provides instructions to the language model. This controls what the language model outputs, so understanding how to construct prompts and different prompting strategies is crucial.
- Output Parsers: These translate the raw response from the LLM to a more workable format, making it easy to use the output downstream.

In this getting started guide we will cover those three components by themselves, and then cover the LLMChain which combines all of them. Understanding these concepts will set you up well for being able to use and customize LangChain applications. Most LangChain applications allow you to configure the LLM and/or the prompt used, so knowing how to take advantage of this will be a big enabler.

LLMs

There are two types of language models, which in LangChain are called:

- LLMs: this is a language model which takes a string as input and returns a string
- ChatModels: this is a language model which takes a list of messages as input and returns a message

The input/output for LLMs is simple and easy to understand - a string. But what about ChatModels? The input there is a list of `ChatMessage`s, and the output is a single `ChatMessage`. A `ChatMessage` has two required components:

- `content`: This is the content of the message.
- `role`: This is the role of the entity from which the `ChatMessage` is coming from.

LangChain provides several objects to easily distinguish between different roles:

- `HumanMessage`: A `ChatMessage` coming from a human/user.
- `AIMessage`: A `ChatMessage` coming from an AI/assistant.
- `SystemMessage`: A `ChatMessage` coming from the system.
- `FunctionMessage`: A `ChatMessage` coming from a function call.

If none of those roles sound right, there is also a `ChatMessage` class where you can specify the role manually. For more information on how to use these different messages most effectively, see our prompting guide.

LangChain exposes a standard interface for both, but it's useful to understand this difference in order to construct prompts for a given language model. The standard interface that LangChain exposes has two methods:

- `predict`: Takes in a string, returns a string
- `predict_messages`: Takes in a list of messages, returns a message.

Let's see how to work with these different types of models and these different types of inputs. First, let's import an LLM and a ChatModel.

```
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI

llm = OpenAI()
chat_model = ChatOpenAI()

llm.predict("hi!")
>>> "Hi"

chat_model.predict("hi!")
>>> "Hi"
```

The `OpenAI` and `ChatOpenAI` objects are basically just configuration objects. You can initialize them with parameters like `temperature` and others, and pass them around.

Next, let's use the `predict` method to run over a string input.

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

```
llm.predict(text)
# >> Feetful of Fun

chat_model.predict(text)
# >> Socks O'Color
```

Finally, let's use the `predict_messages` method to run over a list of messages.

```
from langchain.schema import HumanMessage

text = "What would be a good company name for a company that makes
colorful socks?"
messages = [HumanMessage(content=text)]

llm.predict_messages(messages)
# >> Feetful of Fun

chat_model.predict_messages(messages)
# >> Socks O'Color
```

For both these methods, you can also pass in parameters as key word arguments. For example, you could pass in `temperature=0` to adjust the temperature that is used from what the object was configured with. Whatever values are passed in during run time will always override what the object was configured with.

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.

PromptTemplates help with exactly this! They bundle up all the logic for going from user input into a fully formatted prompt. This can start off very simple - for example, a prompt to produce the above string would just be:

```
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?
```

However, the advantages of using these over raw string formatting are several. You can "partial" out variables - eg you can format only some of the variables at a time. You can compose them together, easily combining different templates into a single prompt. For explanations of these functionalities, see the section on prompts for more detail.

PromptTemplates can also be used to produce a list of messages. In this case, the prompt not only contains information about the content, but also each message (its role, its position in the list, etc) Here, what happens most often is a ChatPromptTemplate is a list of ChatMessageTemplates. Each ChatMessageTemplate contains instructions for how to format that ChatMessage - its role, and then also its content. Let's take a look at this below:

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

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])

chat_prompt.format_messages(input_language="English",
output_language="French", text="I love programming.")
```

```
[
    SystemMessage(content="You are a helpful assistant that translates
```

```
English to French.", additional_kwargs={}),
    HumanMessage(content="I love programming.")
]
```

ChatPromptTemplates can also include other things besides ChatMessageTemplates - see the section on prompts for more detail.

Output Parsers

OutputParsers convert the raw output of an LLM into a format that can be used downstream. There are few main type of OutputParsers, including:

- Convert text from LLM -> structured information (eg JSON)
- Convert a ChatMessage into just a string
- Convert the extra information returned from a call besides the message (like OpenAI function invocation) into a string.

For full information on this, see the section on output parsers

In this getting started guide, we will write our own output parser - one that converts a comma separated list into a list.

```
from langchain.schema import BaseOutputParser

class CommaSeparatedListOutputParser(BaseOutputParser):
    """Parse the output of an LLM call to a comma-separated list."""


    def parse(self, text: str):
        """Parse the output of an LLM call."""
        return text.strip().split(", ")

CommaSeparatedListOutputParser().parse("hi, bye")
# >> ['hi', 'bye']
```

LLMChain

We can now combine all these into one chain. This chain will take input variables, pass those to a prompt template to create a prompt, pass the prompt to an LLM, and then pass the output

through an (optional) output parser. This is a convenient way to bundle up a modular piece of logic. Let's see it in action!

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from langchain.chains import LLMChain
from langchain.schema import BaseOutputParser

class CommaSeparatedListOutputParser(BaseOutputParser):
    """Parse the output of an LLM call to a comma-separated list."""


    def parse(self, text: str):
        """Parse the output of an LLM call."""
        return text.strip().split(", ")

template = """You are a helpful assistant who generates comma separated
lists.
A user will pass in a category, and you should generate 5 objects in
that category in a comma separated list.
ONLY return a comma separated list, and nothing more."""
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=ChatOpenAI(),
    prompt=chat_prompt,
    output_parser=CommaSeparatedListOutputParser()
)
chain.run("colors")
# >> ['red', 'blue', 'green', 'yellow', 'orange']
```

Next Steps

This is it! We've now gone over how to create the core building block of LangChain applications - the LLMChains. There is a lot more nuance in all these components (LLMs, prompts, output parsers) and a lot more different components to learn about as well. To continue on your journey:

- Dive deeper into LLMs, prompts, and output parsers
- Learn the other key components
- Check out our helpful guides for detailed walkthroughs on particular topics
- Explore end-to-end use cases

How to add memory to a Multi-Input Chain

Most memory objects assume a single input. In this notebook, we go over how to add memory to a chain that has multiple inputs. As an example of such a chain, we will add memory to a question/answering chain. This chain takes as inputs both related documents and a user question.

```
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.embeddings.cohere import CohereEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores.elastic_vector_search import
ElasticVectorSearch
from langchain.vectorstores import Chroma
from langchain.docstore.document import Document
```

API Reference:

- OpenAIEmbeddings from `langchain.embeddings.openai`
- CohereEmbeddings from `langchain.embeddings.cohere`
- CharacterTextSplitter from `langchain.text_splitter`
- ElasticVectorSearch from `langchain.vectorstores.elastic_vector_search`
- Chroma from `langchain.vectorstores`
- Document from `langchain.docstore.document`

```
with open("../../state_of_the_union.txt") as f:
    state_of_the_union = f.read()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_text(state_of_the_union)

embeddings = OpenAIEmbeddings()
```

```
docsearch = Chroma.from_texts(
    texts, embeddings, metadatas=[{"source": i} for i in
range(len(texts))]
)
```

```
    Running Chroma using direct local API.
    Using DuckDB in-memory for database. Data will be transient.
```

```
query = "What did the president say about Justice Breyer"
docs = docsearch.similarity_search(query)
```

```
from langchain.chains.question_answering import load_qa_chain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.memory import ConversationBufferMemory
```

API Reference:

- load_qa_chain from `langchain.chains.question_answering`
- OpenAI from `langchain.llms`
- PromptTemplate from `langchain.prompts`
- ConversationBufferMemory from `langchain.memory`

```
template = """You are a chatbot having a conversation with a human.

Given the following extracted parts of a long document and a question,
create a final answer.

{context}

{chat_history}
Human: {human_input}
Chatbot:"""

prompt = PromptTemplate(
    input_variables=["chat_history", "human_input", "context"],
template=template
)
memory = ConversationBufferMemory(memory_key="chat_history",
input_key="human_input")
chain = load_qa_chain(
    OpenAI(temperature=0), chain_type="stuff", memory=memory,
prompt=prompt
)
```

```
query = "What did the president say about Justice Breyer"
chain({"input_documents": docs, "human_input": query},
return_only_outputs=True)
```

```
    {'output_text': ' Tonight, I'd like to honor someone who has
dedicated his life to serve this country: Justice Stephen Breyer—an
Army veteran, Constitutional scholar, and retiring Justice of the
United States Supreme Court. Justice Breyer, thank you for your
service.'}
```

```
print(chain.memory.buffer)
```

```
    Human: What did the president say about Justice Breyer
    AI:  Tonight, I'd like to honor someone who has dedicated his life
to serve this country: Justice Stephen Breyer—an Army veteran,
Constitutional scholar, and retiring Justice of the United States
Supreme Court. Justice Breyer, thank you for your service.
```

How to add Memory to an Agent

This notebook goes over adding memory to an Agent. Before going through this notebook, please walkthrough the following notebooks, as this will build on top of both of them:

- Adding memory to an LLM Chain
- Custom Agents

In order to add a memory to an agent we are going to the the following steps:

1. We are going to create an LLMChain with memory.
2. We are going to use that LLMChain to create a custom Agent.

For the purposes of this exercise, we are going to create a simple custom Agent that has access to a search tool and utilizes the `ConversationBufferMemory` class.

```
from langchain.agents import ZeroShotAgent, Tool, AgentExecutor
from langchain.memory import ConversationBufferMemory
from langchain import OpenAI, LLMChain
from langchain.utilities import GoogleSearchAPIWrapper
```

API Reference:

- ZeroShotAgent from `langchain.agents`
- Tool from `langchain.agents`
- AgentExecutor from `langchain.agents`
- ConversationBufferMemory from `langchain.memory`
- GoogleSearchAPIWrapper from `langchain.utilities`

```
search = GoogleSearchAPIWrapper()
tools = [
    Tool(
        name="Search",
        func=search.run,
        description="useful for when you need to answer questions about
current events",
```

```
    )
]
```

Notice the usage of the `chat_history` variable in the PromptTemplate, which matches up with the dynamic key name in the ConversationBufferMemory.

```
prefix = """Have a conversation with a human, answering the following
questions as best you can. You have access to the following tools:"""
suffix = """Begin!"

{chat_history}
Question: {input}
{agent_scratchpad}"""

prompt = ZeroShotAgent.create_prompt(
    tools,
    prefix=prefix,
    suffix=suffix,
    input_variables=["input", "chat_history", "agent_scratchpad"],
)
memory = ConversationBufferMemory(memory_key="chat_history")
```

We can now construct the LLMChain, with the Memory object, and then create the agent.

```
llm_chain = LLMChain(llm=OpenAI(temperature=0), prompt=prompt)
agent = ZeroShotAgent(llm_chain=llm_chain, tools=tools, verbose=True)
agent_chain = AgentExecutor.from_agent_and_tools(
    agent=agent, tools=tools, verbose=True, memory=memory
)
```

```
agent_chain.run(input="How many people live in canada?")
```

```
    > Entering new AgentExecutor chain...
    Thought: I need to find out the population of Canada
    Action: Search
    Action Input: Population of Canada
    Observation: The current population of Canada is 38,566,192 as of
Saturday, December 31, 2022, based on Worldometer elaboration of the
latest United Nations data. · Canada ... Additional information related
```

```
to Canadian population trends can be found on Statistics Canada's
Population and Demography Portal. Population of Canada (real- ... Index
to the latest information from the Census of Population. This survey
conducted by Statistics Canada provides a statistical portrait of
Canada and its ... 14 records ... Estimated number of persons by
quarter of a year and by year, Canada, provinces and territories. The
2021 Canadian census counted a total population of 36,991,981, an
increase of around 5.2 percent over the 2016 figure. ... Between 1990
and 2008, the ... ( 2 ) Census reports and other statistical
publications from national statistical offices, ( 3 ) Eurostat:
Demographic Statistics, ( 4 ) United Nations ... Canada is a country in
North America. Its ten provinces and three territories extend from ...
Population. • Q4 2022 estimate. 39,292,355 (37th). Information is
available for the total Indigenous population and each of the three ...
The term 'Aboriginal' or 'Indigenous' used on the Statistics Canada ...
Jun 14, 2022 ... Determinants of health are the broad range of
personal, social, economic and environmental factors that determine
individual and population ... COVID-19 vaccination coverage across
Canada by demographics and key populations. Updated every Friday at
12:00 PM Eastern Time.
    Thought: I now know the final answer
    Final Answer: The current population of Canada is 38,566,192 as of
Saturday, December 31, 2022, based on Worldometer elaboration of the
latest United Nations data.
    > Finished AgentExecutor chain.
```

```
    'The current population of Canada is 38,566,192 as of Saturday,
December 31, 2022, based on Worldometer elaboration of the latest
United Nations data.'
```

To test the memory of this agent, we can ask a followup question that relies on information in the previous exchange to be answered correctly.

```
agent_chain.run(input="what is their national anthem called?")
```

```
    > Entering new AgentExecutor chain...
    Thought: I need to find out what the national anthem of Canada is
called.
    Action: Search
```

```
    Action Input: National Anthem of Canada
    Observation: Jun 7, 2010 ...
https://twitter.com/CanadaImmigrantCanadian National Anthem O Canada in
HQ - complete with lyrics, captions, vocals & music.LYRICS:O Canada!
Nov 23, 2022 ... After 100 years of tradition, O Canada was proclaimed
Canada's national anthem in 1980. The music for O Canada was composed
in 1880 by Calixa ... O Canada, national anthem of Canada. It was
proclaimed the official national anthem on July 1, 1980. “God Save the
Queen” remains the royal anthem of Canada ... O Canada! Our home and
native land! True patriot love in all of us command. Car ton bras sait
porter l'épée,. Il sait porter la croix! "O Canada" (French: Ô Canada)
is the national anthem of Canada. The song was originally commissioned
by Lieutenant Governor of Quebec Théodore Robitaille ... Feb 1, 2018
... It was a simple tweak – just two words. But with that, Canada just
voted to make its national anthem, “O Canada,” gender neutral, ... "O
Canada" was proclaimed Canada's national anthem on July 1,. 1980, 100
years after it was first sung on June 24, 1880. The music. Patriotic
music in Canada dates back over 200 years as a distinct category from
British or French patriotism, preceding the first legal steps to ...
Feb 4, 2022 ... English version: O Canada! Our home and native land!
True patriot love in all of us command. With glowing hearts we ... Feb
1, 2018 ... Canada's Senate has passed a bill making the country's
national anthem gender-neutral. If you're not familiar with the words
to “O Canada,” ...
    Thought: I now know the final answer.
    Final Answer: The national anthem of Canada is called "O Canada".
    > Finished AgentExecutor chain.



    'The national anthem of Canada is called "O Canada".'
```

We can see that the agent remembered that the previous question was about Canada, and properly asked Google Search what the name of Canada's national anthem was.

For fun, let's compare this to an agent that does NOT have memory.

```
prefix = """Have a conversation with a human, answering the following
questions as best you can. You have access to the following tools:"""
suffix = """Begin!"

Question: {input}
{agent_scratchpad}"""
```

```
prompt = ZeroShotAgent.create_prompt(
    tools, prefix=prefix, suffix=suffix, input_variables=["input",
"agent_scratchpad"]
)
llm_chain = LLMChain(llm=OpenAI(temperature=0), prompt=prompt)
agent = ZeroShotAgent(llm_chain=llm_chain, tools=tools, verbose=True)
agent_without_memory = AgentExecutor.from_agent_and_tools(
    agent=agent, tools=tools, verbose=True
)
```

```
agent_without_memory.run("How many people live in canada?")
```

```
    > Entering new AgentExecutor chain...
    Thought: I need to find out the population of Canada
    Action: Search
    Action Input: Population of Canada
    Observation: The current population of Canada is 38,566,192 as of
Saturday, December 31, 2022, based on Worldometer elaboration of the
latest United Nations data. · Canada ... Additional information related
to Canadian population trends can be found on Statistics Canada's
Population and Demography Portal. Population of Canada (real- ... Index
to the latest information from the Census of Population. This survey
conducted by Statistics Canada provides a statistical portrait of
Canada and its ... 14 records ... Estimated number of persons by
quarter of a year and by year, Canada, provinces and territories. The
2021 Canadian census counted a total population of 36,991,981, an
increase of around 5.2 percent over the 2016 figure. ... Between 1990
and 2008, the ... ( 2 ) Census reports and other statistical
publications from national statistical offices, ( 3 ) Eurostat:
Demographic Statistics, ( 4 ) United Nations ... Canada is a country in
North America. Its ten provinces and three territories extend from ...
Population. • Q4 2022 estimate. 39,292,355 (37th). Information is
available for the total Indigenous population and each of the three ...
The term 'Aboriginal' or 'Indigenous' used on the Statistics Canada ...
Jun 14, 2022 ... Determinants of health are the broad range of
personal, social, economic and environmental factors that determine
individual and population ... COVID-19 vaccination coverage across
Canada by demographics and key populations. Updated every Friday at
12:00 PM Eastern Time.
    Thought: I now know the final answer
    Final Answer: The current population of Canada is 38,566,192 as of
Saturday, December 31, 2022, based on Worldometer elaboration of the
```

```
latest United Nations data.
    > Finished AgentExecutor chain.
```

```
    'The current population of Canada is 38,566,192 as of Saturday,
December 31, 2022, based on Worldometer elaboration of the latest
United Nations data.'
```

```
agent_without_memory.run("what is their national anthem called?")
```

```
    > Entering new AgentExecutor chain...
    Thought: I should look up the answer
    Action: Search
    Action Input: national anthem of [country]
    Observation: Most nation states have an anthem, defined as "a song,
as of praise, devotion, or patriotism"; most anthems are either marches
or hymns in style. List of all countries around the world with its
national anthem. ... Title and lyrics in the language of the country
and translated into English, Aug 1, 2021 ... 1. Afghanistan, "Milli
Surood" (National Anthem) · 2. Armenia, "Mer Hayrenik" (Our Fatherland)
· 3. Azerbaijan (a transcontinental country with ... A national anthem
is a patriotic musical composition symbolizing and evoking eulogies of
the history and traditions of a country or nation. National Anthem of
Every Country ; Fiji, “Meda Dau Doka” (“God Bless Fiji”) ; Finland,
“Maamme”. (“Our Land”) ; France, “La Marseillaise” (“The
Marseillaise”). You can find an anthem in the menu at the top
alphabetically or you can use the search feature. This site is focussed
on the scholarly study of national anthems ... Feb 13, 2022 ... The 38-
year-old country music artist had the honor of singing the National
Anthem during this year's big game, and she did not disappoint. Oldest
of the World's National Anthems ; France, La Marseillaise (“The
Marseillaise”), 1795 ; Argentina, Himno Nacional Argentino (“Argentine
National Anthem”) ... Mar 3, 2022 ... Country music star Jessie James
Decker gained the respect of music and hockey fans alike after a jaw-
dropping rendition of "The Star-Spangled ... This list shows the
country on the left, the national anthem in the ... There are many
countries over the world who have a national anthem of their own.
    Thought: I now know the final answer
    Final Answer: The national anthem of [country] is [name of anthem].
    > Finished AgentExecutor chain.
```

```
'The national anthem of [country] is [name of anthem].'
```

Agents

The core idea of agents is to use an LLM to choose a sequence of actions to take. In chains, a sequence of actions is hardcoded (in code). In agents, a language model is used as a reasoning engine to determine which actions to take and in which order.

There are several key components here:

Agent

This is the class responsible for deciding what step to take next. This is powered by a language model and a prompt. This prompt can include things like:

1. The personality of the agent (useful for having it respond in a certain way)
2. Background context for the agent (useful for giving it more context on the types of tasks it's being asked to do)
3. Prompting strategies to invoke better reasoning (the most famous/widely used being ReAct)

LangChain provides a few different types of agents to get started. Even then, you will likely want to customize those agents with parts (1) and (2). For a full list of agent types see agent types

Tools

Tools are functions that an agent calls. There are two important considerations here:

1. Giving the agent access to the right tools
2. Describing the tools in a way that is most helpful to the agent

Without both, the agent you are trying to build will not work. If you don't give the agent access to a correct set of tools, it will never be able to accomplish the objective. If you don't describe the tools properly, the agent won't know how to properly use them.

LangChain provides a wide set of tools to get started, but also makes it easy to define your own (including custom descriptions). For a full list of tools, see here

Toolkits

Often the set of tools an agent has access to is more important than a single tool. For this LangChain provides the concept of toolkits - groups of tools needed to accomplish specific objectives. There are generally around 3-5 tools in a toolkit.

LangChain provides a wide set of toolkits to get started. For a full list of toolkits, see here

AgentExecutor

The agent executor is the runtime for an agent. This is what actually calls the agent and executes the actions it chooses. Pseudocode for this runtime is below:

```
next_action = agent.get_action(...)
while next_action != AgentFinish:
    observation = run(next_action)
    next_action = agent.get_action(..., next_action, observation)
return next_action
```

While this may seem simple, there are several complexities this runtime handles for you, including:

1. Handling cases where the agent selects a non-existent tool
2. Handling cases where the tool errors
3. Handling cases where the agent produces output that cannot be parsed into a tool invocation
4. Logging and observability at all levels (agent decisions, tool calls) either to stdout or LangSmith.

Other types of agent runtimes

The `AgentExecutor` class is the main agent runtime supported by LangChain. However, there are other, more experimental runtimes we also support. These include:

- Plan-and-execute Agent
- Baby AGI
- Auto GPT

Get started

This will go over how to get started building an agent. We will use a LangChain agent class, but show how to customize it to give it specific context. We will then define custom tools, and then run it all in the standard LangChain AgentExecutor.

Set up the agent

We will use the OpenAIFunctionsAgent. This is easiest and best agent to get started with. It does however require usage of ChatOpenAI models. If you want to use a different language model, we would recommend using the ReAct agent.

For this guide, we will construct a custom agent that has access to a custom tool. We are choosing this example because we think for most use cases you will NEED to customize either the agent or the tools. The tool we will give the agent is a tool to calculate the length of a word. This is useful because this is actually something LLMs can mess up due to tokenization. We will first create it WITHOUT memory, but we will then show how to add memory in. Memory is needed to enable conversation.

First, let's load the language model we're going to use to control the agent.

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(temperature=0)
```

Next, let's define some tools to use. Let's write a really simple Python function to calculate the length of a word that is passed in.

```
from langchain.agents import tool

@tool
def get_word_length(word: str) -> int:
    """Returns the length of a word."""
    return len(word)

tools = [get_word_length]
```

Now let us create the prompt. We can use the `OpenAIFunctionsAgent.create_prompt` helper function to create a prompt automatically. This allows for a few different ways to customize, including passing in a custom SystemMessage, which we will do.

```
from langchain.schema import SystemMessage
from langchain.agents import OpenAIFunctionsAgent
system_message = SystemMessage(content="You are very powerful
assistant, but bad at calculating lengths of words.")
prompt =
OpenAIFunctionsAgent.create_prompt(system_message=system_message)
```

Putting those pieces together, we can now create the agent.

```
agent = OpenAIFunctionsAgent(llm=llm, tools=tools, prompt=prompt)
```

Finally, we create the AgentExecutor - the runtime for our agent.

```
from langchain.agents import AgentExecutor
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

Now let's test it out!

```
agent_executor.run("how many letters in the word educa?")
```

```
> Entering new AgentExecutor chain...

Invoking: `get_word_length` with `{'word': 'educa'}`

5

There are 5 letters in the word "educa".

> Finished chain.

'There are 5 letters in the word "educa".'
```

This is great - we have an agent! However, this agent is stateless - it doesn't remember anything about previous interactions. This means you can't ask follow up questions easily. Let's fix that by adding in memory.

In order to do this, we need to do two things:

1. Add a place for memory variables to go in the prompt
2. Add memory to the AgentExecutor (note that we add it here, and NOT to the agent, as this is the outermost chain)

First, let's add a place for memory in the prompt. We do this by adding a placeholder for messages with the key `"chat_history"`.

```
from langchain.prompts import MessagesPlaceholder

MEMORY_KEY = "chat_history"
prompt = OpenAIFunctionsAgent.create_prompt(
    system_message=system_message,
    extra_prompt_messages=
[MessagesPlaceholder(variable_name=MEMORY_KEY)]
)
```

Next, let's create a memory object. We will do this by using `ConversationBufferMemory`. Importantly, we set `memory_key` also equal to `"chat_history"` (to align it with the prompt) and set `return_messages` (to make it return messages rather than a string).

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(memory_key=MEMORY_KEY,
return_messages=True)
```

We can then put it all together!

```
agent = OpenAIFunctionsAgent(llm=llm, tools=tools, prompt=prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, memory=memory,
verbose=True)
agent_executor.run("how many letters in the word educa?")
agent_executor.run("is that a real word?")
```

Agent types

Action agents

Agents 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 a response to the user. Here are the agents available in LangChain.

Zero-shot ReAct

This agent uses the ReAct framework to determine which tool to use based solely on the tool's description. Any number of tools can be provided. This agent requires that a description is provided for each tool.

Note: This is the most general purpose action agent.

Structured input ReAct

The structured tool chat agent is capable of using multi-input tools. Older agents are configured to specify an action input as a single string, but this agent can use a tools' argument schema to create a structured action input. This is useful for more complex tool usage, like precisely navigating around a browser.

OpenAI Functions

Certain OpenAI models (like gpt-3.5-turbo-0613 and gpt-4-0613) have been explicitly fine-tuned to detect when a function should be called and respond with the inputs that should be passed to the function. The OpenAI Functions Agent is designed to work with these models.

Conversational

This agent is designed to be used in conversational settings. The prompt is designed to make the agent helpful and conversational. It uses the ReAct framework to decide which tool to use, and uses memory to remember the previous conversation interactions.

Self ask with search

This agent utilizes a single tool that should be named `Intermediate Answer`. This tool should be able to lookup factual answers to questions. This agent is equivalent to the original

self ask with search paper, where a Google search API was provided as the tool.

ReAct document store

This agent uses the ReAct framework to interact with a docstore. Two tools must be provided: a `Search` tool and a `Lookup` tool (they must be named exactly as so). The `Search` tool should search for a document, while the `Lookup` tool should lookup a term in the most recently found document. This agent is equivalent to the original ReAct paper, specifically the Wikipedia example.

Plan-and-execute agents

Plan and execute agents accomplish an objective by first planning what to do, then executing the sub tasks. This idea is largely inspired by BabyAGI and then the "Plan-and-Solve" paper.

Prompt templates

Prompt templates are pre-defined recipes for generating prompts for language models.

A template may include instructions, few shot examples, and specific context and questions appropriate for a given task.

LangChain provides tooling to create and work with prompt templates.

LangChain strives to create model agnostic templates to make it easy to reuse existing templates across different language models.

Typically, language models expect the prompt to either be a string or else a list of chat messages.

Prompt template

Use `PromptTemplate` to create a template for a string prompt.

By default, `PromptTemplate` uses Python's str.format syntax for templating; however other templating syntax is available (e.g., `jinja2`).

```
from langchain import PromptTemplate

prompt_template = PromptTemplate.from_template(
    "Tell me a {adjective} joke about {content}."
)
prompt_template.format(adjective="funny", content="chickens")
```

```
"Tell me a funny joke about chickens."
```

The template supports any number of variables, including no variables:

```
from langchain import PromptTemplate

prompt_template = PromptTemplate.from_template(
"Tell me a joke"
```

```
)
prompt_template.format()
```

For additional validation, specify `input_variables` explicitly. These variables will be compared against the variables present in the template string during instantiation, raising an exception if there is a mismatch; for example,

```
from langchain import PromptTemplate

invalid_prompt = PromptTemplate(
    input_variables=["adjective"],
    template="Tell me a {adjective} joke about {content}."
)
```

You can create custom prompt templates that format the prompt in any way you want. For more information, see Custom Prompt Templates.

Chat prompt template

The prompt to Chat Models is a list of chat messages.

Each chat message is associated with content, and an additional parameter called `role`. For example, in the OpenAI Chat Completions API, a chat message can be associated with an AI assistant, a human or a system role.

Create a chat prompt template like this:

```
from langchain.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful AI bot. Your name is {name}."),
    ("human", "Hello, how are you doing?"),
    ("ai", "I'm doing well, thanks!"),
    ("human", "{user_input}"),
])

messages = template.format_messages(
    name="Bob",
    user_input="What is your name?"
)
```

`ChatPromptTemplate.from_messages` accepts a variety of message representations.

For example, in addition to using the 2-tuple representation of (type, content) used above, you could pass in an instance of `MessagePromptTemplate` or `BaseMessage`.

```
from langchain.prompts import ChatPromptTemplate
from langchain.prompts.chat import SystemMessage,
HumanMessagePromptTemplate

template = ChatPromptTemplate.from_messages(
    [
        SystemMessage(
            content=(
                "You are a helpful assistant that re-writes the user's
text to "
                "sound more upbeat."
            )
        ),
        HumanMessagePromptTemplate.from_template("{text}"),
    ]
)

from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI()
llm(template.format_messages(text='i dont like eating tasty things.'))
```

```
AIMessage(content='I absolutely adore indulging in delicious treats!',
additional_kwargs={}, example=False)
```

This provides you with a lot of flexibility in how you construct your chat prompts.

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?

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.

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

PROMPT = """\
Given the function name and source code, generate an English language
explanation of the function.
Function Name: {function_name}
Source Code:
{source_code}
Explanation:
"""


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 = PROMPT.format(
            function_name=kwargs["function_name"].__name__,
source_code=source_code
        )
```

```
        return prompt

    def _prompt_type(self):
        return "function-explainer"
```

API Reference:

- StringPromptTemplate from `langchain.prompts`

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

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

Using an example set

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

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`

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

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`

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, and the optimal prompt compilation will likely vary by model. Because of this, we provide few-shot prompt templates like the FewShotChatMessagePromptTemplate as a flexible starting point, and you can modify or replace them as you see fit.

The goal of few-shot prompt templates are to dynamically select examples based on an input, and then format the examples in a final prompt to provide for the model.

Note: The following code examples are for chat models. For similar few-shot prompt examples for completion models (LLMs), see the few-shot prompt templates guide.

Fixed Examples

The most basic (and common) few-shot prompting technique is to use a fixed prompt example. This way you can select a chain, evaluate it, and avoid worrying about additional moving parts in production.

The basic components of the template are:

- `examples`: A list of dictionary examples to include in the final prompt.
- `example_prompt`: converts each example into 1 or more messages through its `format_messages` method. A common example would be to convert each example into one human message and one AI message response, or a human message followed by a function call message.

Below is a simple demonstration. First, import the modules for this example:

```
from langchain.prompts import (
    FewShotChatMessagePromptTemplate,
    ChatPromptTemplate,
)
```

API Reference:

- FewShotChatMessagePromptTemplate from `langchain.prompts`

- ChatPromptTemplate from `langchain.prompts`

Then, define the examples you'd like to include.

```
examples = [
    {"input": "2+2", "output": "4"},
    {"input": "2+3", "output": "5"},
]
```

Next, assemble them into the few-shot prompt template.

```
# This is a prompt template used to format each individual example.
example_prompt = ChatPromptTemplate.from_messages(
    [
        ("human", "{input}"),
        ("ai", "{output}"),
    ]
)
few_shot_prompt = FewShotChatMessagePromptTemplate(
    example_prompt=example_prompt,
    examples=examples,
)

print(few_shot_prompt.format())
```

```
    Human: 2+2
    AI: 4
    Human: 2+3
    AI: 5
```

Finally, assemble your final prompt and use it with a model.

```
final_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are wonderous wizard of math."),
        few_shot_prompt,
        ("human", "{input}"),
    ]
)
```

```
from langchain.chat_models import ChatAnthropic

chain = final_prompt | ChatAnthropic(temperature=0.0)

chain.invoke({"input": "What's the square of a triangle?"})
```

API Reference:

- ChatAnthropic from `langchain.chat_models`

```
    AIMessage(content=' Triangles do not have a "square". A square
refers to a shape with 4 equal sides and 4 right angles. Triangles have
3 sides and 3 angles.\n\nThe area of a triangle can be calculated using
the formula:\n\nA = 1/2 * b * h\n\nWhere:\n\nA is the area \nb is the
base (the length of one of the sides)\nh is the height (the length from
the base to the opposite vertex)\n\nSo the area depends on the specific
dimensions of the triangle. There is no single "square of a triangle".
The area can vary greatly depending on the base and height
measurements.', additional_kwargs={}, example=False)
```

Dynamic Few-shot Prompting

Sometimes you may want to condition which examples are shown based on the input. For this, you can replace the `examples` with an `example_selector`. The other components remain the same as above! To review, the dynamic few-shot prompt template would look like:

- `example_selector`: responsible for selecting few-shot examples (and the order in which they are returned) for a given input. These implement the BaseExampleSelector interface. A common example is the vectorstore-backed SemanticSimilarityExampleSelector
- `example_prompt`: convert each example into 1 or more messages through its `format_messages` method. A common example would be to convert each example into one human message and one AI message response, or a human message followed by a function call message.

These once again can be composed with other messages and chat templates to assemble your final prompt.

```
from langchain.prompts import SemanticSimilarityExampleSelector
from langchain.embeddings import OpenAIEmbeddings
```

```
from langchain.vectorstores import Chroma
```

API Reference:

- SemanticSimilarityExampleSelector from `langchain.prompts`
- OpenAIEmbeddings from `langchain.embeddings`
- Chroma from `langchain.vectorstores`

Since we are using a vectorstore to select examples based on semantic similarity, we will want to first populate the store.

```
examples = [
    {"input": "2+2", "output": "4"},
    {"input": "2+3", "output": "5"},
    {"input": "2+4", "output": "6"},
    {"input": "What did the cow say to the moon?", "output": "nothing
at all"},
    {
        "input": "Write me a poem about the moon",
        "output": "One for the moon, and one for me, who are we to talk
about the moon?",
    },
]

to_vectorize = [" ".join(example.values()) for example in examples]
embeddings = OpenAIEmbeddings()
vectorstore = Chroma.from_texts(to_vectorize, embeddings,
metadatas=examples)
```

Create the `example_selector`

With a vectorstore created, you can create the `example_selector`. Here we will isntruct it to only fetch the top 2 examples.

```
example_selector = SemanticSimilarityExampleSelector(
    vectorstore=vectorstore,
    k=2,
)

# The prompt template will load examples by passing the input do the
`select_examples` method
example_selector.select_examples({"input": "horse"})
```

```
    [{'input': 'What did the cow say to the moon?', 'output': 'nothing
at all'},
     {'input': '2+4', 'output': '6'}]
```

Create prompt template

Assemble the prompt template, using the `example_selector` created above.

```
from langchain.prompts import (
    FewShotChatMessagePromptTemplate,
    ChatPromptTemplate,
)

# Define the few-shot prompt.
few_shot_prompt = FewShotChatMessagePromptTemplate(
    # The input variables select the values to pass to the
example_selector
    input_variables=["input"],
    example_selector=example_selector,
    # Define how each example will be formatted.
    # In this case, each example will become 2 messages:
    # 1 human, and 1 AI
    example_prompt=ChatPromptTemplate.from_messages(
        [("human", "{input}"), ("ai", "{output}")]
    ),
)
```

API Reference:

- FewShotChatMessagePromptTemplate from `langchain.prompts`
- ChatPromptTemplate from `langchain.prompts`

Below is an example of how this would be assembled.

```
print(few_shot_prompt.format(input="What's 3+3?"))
```

```
    Human: 2+3
    AI: 5
    Human: 2+2
    AI: 4
```

Assemble the final prompt template:

```
final_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are wonderous wizard of math."),
        few_shot_prompt,
        ("human", "{input}"),
    ]
)
```

```
print(few_shot_prompt.format(input="What's 3+3?"))
```

```
    Human: 2+3
    AI: 5
    Human: 2+2
    AI: 4
```

Use with an LLM

Now, you can connect your model to the few-shot prompt.

```
from langchain.chat_models import ChatAnthropic

chain = final_prompt | ChatAnthropic(temperature=0.0)

chain.invoke({"input": "What's 3+3?"})
```

API Reference:

- ChatAnthropic from `langchain.chat_models`

```
    AIMessage(content=' 3 + 3 = 6', additional_kwargs={},
example=False)
```

Format template output

The output of the format method is available as string, list of messages and `ChatPromptValue`

As string:

```
output = chat_prompt.format(input_language="English",
output_language="French", text="I love programming.")
output
```

```
    'System: You are a helpful assistant that translates English to
French.\nHuman: I love programming.'
```

```
# or alternatively
output_2 = chat_prompt.format_prompt(input_language="English",
output_language="French", text="I love programming.").to_string()

assert output == output_2
```

As `ChatPromptValue`

```
chat_prompt.format_prompt(input_language="English",
output_language="French", text="I love programming.")
```

```
    ChatPromptValue(messages=[SystemMessage(content='You are a helpful
assistant that translates English to French.', additional_kwargs={}),
HumanMessage(content='I love programming.', additional_kwargs={})])
```

As list of Message objects

```
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={})]
```

LLMs

> **INFO**
>
> Head to Integrations for documentation on built-in integrations with LLM providers.

Large Language Models (LLMs) are a core component of LangChain. LangChain does not serve its own LLMs, but rather provides a standard interface for interacting with many different LLMs.

Get started

There are lots of LLM providers (OpenAI, Cohere, Hugging Face, etc) - the `LLM` class is designed to provide a standard interface for all of them.

In this walkthrough we'll work with an OpenAI LLM wrapper, although the functionalities highlighted are generic for all LLM types.

Setup

To start we'll need to install the OpenAI 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="...")
```

otherwise you can initialize without any params:

```
from langchain.llms import OpenAI

llm = OpenAI()
```

`__call__`: string in -> string out

The simplest way to use an LLM is a callable: pass in a string, get a string completion.

```
llm("Tell me a joke")
```

```
    'Why did the chicken cross the road?\n\nTo get to the other side.'
```

`generate`: batch calls, richer outputs

`generate` lets you can call the model with a list of strings, getting back a more complete response than just the text. This complete response can include things like multiple top responses and other LLM provider-specific information:

```
llm_result = llm.generate(["Tell me a joke", "Tell me a poem"]*15)
```

```
len(llm_result.generations)
```

```
    30
```

```
llm_result.generations[0]
```

```
    [Generation(text='\n\nWhy did the chicken cross the road?\n\nTo get
to the other side!'),
     Generation(text='\n\nWhy did the chicken cross the road?\n\nTo get
to the other side.')]
```

```
llm_result.generations[-1]
```

```
    [Generation(text="\n\nWhat if love neverspeech\n\nWhat if love
never ended\n\nWhat if love was only a feeling\n\nI'll never know this
love\n\nIt's not a feeling\n\nBut it's what we have for each
other\n\nWe just know that love is something strong\n\nAnd we can't
help but be happy\n\nWe just feel what love is for us\n\nAnd we love
each other with all our heart\n\nWe just don't know how\n\nHow it will
go\n\nBut we know that love is something strong\n\nAnd we'll always
have each other\n\nIn our lives."),
     Generation(text='\n\nOnce upon a time\n\nThere was a love so pure
and true\n\nIt lasted for centuries\n\nAnd never became stale or
dry\n\nIt was moving and alive\n\nAnd the heart of the love-ick\n\nIs
still beating strong and true.')]
```

You can also access provider specific information that is returned. This information is NOT standardized across providers.

```
llm_result.llm_output
```

```
    {'token_usage': {'completion_tokens': 3903,
      'total_tokens': 4023,
      'prompt_tokens': 120}}
```

Chat models

> **INFO**
>
> Head to Integrations for documentation on built-in integrations with chat model providers.

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 started

Setup

To start we'll need to install the OpenAI 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.chat_models import ChatOpenAI

chat = ChatOpenAI(openai_api_key="...")
```

otherwise you can initialize without any params:

```
from langchain.chat_models import ChatOpenAI

chat = ChatOpenAI()
```

Messages

The chat model interface is based around messages rather than raw text. 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`

`__call__`

Messages in -> message out

You can get chat completions by passing one or more messages to the chat model. The response will be a message.

```
from langchain.schema import (
    AIMessage,
    HumanMessage,
    SystemMessage
)

chat([HumanMessage(content="Translate this sentence from English to
French: I love programming.")])
```

```
    AIMessage(content="J'aime programmer.", additional_kwargs={})
```

OpenAI's chat model supports multiple messages as input. See here for more information. Here is an example of sending a system and user message to the chat model:

```
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={})
```

`generate`

Batch calls, richer outputs

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': 77}}
```