# LangChain Cookbook □□□□

*This cookbook is based off the [LangChain Conceptual Documentation]*

**Goal:** Provide an introductory understanding of the components and use cases of LangChain via [ELI5] examples and code snippets. For use cases check out [part 2]. See [video tutorial] of this notebook.

**Links:**

- [LC Conceptual Documentation]
- [LC Python Documentation]
- [LC Javascript/Typescript Documentation]
- [LC Discord]
- [www.langchain.com]
- [LC Twitter]

### What is LangChain?

> LangChain is a framework for developing applications powered by language models.

**~~TL~~DR**: LangChain makes the complicated parts of working & building with AI models easier. It helps do this in two ways:

1. **Integration** - Bring external data, such as your files, other applications, and api data, to your LLMs
2. **Agency** - Allow your LLMs to interact with it's environment via decision making. Use LLMs to help decide which action to take next

### Why LangChain?

1. **Components** - LangChain makes it easy to swap out abstractions and components necessary to work with language models.
2. **Customized Chains** - LangChain provides out of the box support for using and customizing 'chains' - a series of actions strung together.
3. **Speed** □ - This team ships insanely fast. You'll be up to date with the latest LLM features.
4. **Community** □ - Wonderful discord and community support, meet ups, hackathons, etc.

Though LLMs can be straightforward (text-in, text-out) you'll quickly run into friction points that LangChain helps with once you develop more complicated applications.

*Note: This cookbook will not cover all aspects of LangChain. It's contents have been curated to get you to building & impact as quick as possible. For more, please check out [LangChain Conceptual Documentation]*

*Update Oct '23: This notebook has been expanded from it's original form*

You'll need an OpenAI api key to follow this tutorial. You can have it as an environement variable, in an .env file where this jupyter notebook lives, or insert it below where 'YourAPIKey' is. Have if you have questions on this, put these instructions into [ChatGPT].

In [1]:

```
from dotenv import load_dotenv
import os

load_dotenv()

openai_api_key=os.getenv('OPENAI_API_KEY', 'YourAPIKey')
```

# LangChain Components

# Schema - Nuts and Bolts of working with Large Language Models (LLMs)

## Text

**The natural language way to interact with LLMs**

In [2]:

```
# You'll be working with simple strings (that'll soon grow in complexity!)
my_text = "What day comes after Friday?"
my_text
```

Out[2]:

```
'What day comes after Friday?'
```

## Chat Messages

**Like text, but specified with a message type (System, Human, AI)**

- **System - Helpful background context that tell the AI what to do**
- **Human - Messages that are intented to represent the user**
- **AI - Messages that show what the AI responded with**

**For more, see OpenAI's documentation**

In [3]:

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage, AIMessage

# This it the language model we'll use. We'll talk about what we're doing below in the ne
xt section
chat = ChatOpenAI(temperature=.7, openai_api_key=openai_api_key)
```

**Now let's create a few messages that simulate a chat experience with a bot**

In [4]:

```
chat(
    [
        SystemMessage(content="You are a nice AI bot that helps a user figure out what to
eat in one short sentence"),
        HumanMessage(content="I like tomatoes, what should I eat?")
    ]
)
```

Out[4]:

```
AIMessage(content='You could try a caprese salad with fresh tomatoes, mozzarella, and bas
il.')
```

**You can also pass more chat history w/ responses from the AI**

In [5]:

```
chat(
    [
        SystemMessage(content="You are a nice AI bot that helps a user figure out where
to travel in one short sentence"),
        HumanMessage(content="I like the beaches where should I go?"),
        AIMessage(content="You should go to Nice, France"),
        HumanMessage(content="What else should I do when I'm there?")
    ]
)
```

Out[5]:

```
AIMessage(content='You should also explore the charming streets of the Old Town and indul
ge in delicious French cuisine.')
```

**You can also exclude the system message if you want**

In [6]:

```
chat(
    [
        HumanMessage(content="What day comes after Thursday?")
    ]
)
```

Out[6]:

```
AIMessage(content='Friday')
```

### Documents

**An object that holds a piece of text and metadata (more information about that text)**

In [7]:

```
from langchain.schema import Document
```

In [8]:

```
Document(page_content="This is my document. It is full of text that I've gathered from ot
her places",
         metadata={
             'my_document_id' : 234234,
             'my_document_source' : "The LangChain Papers",
             'my_document_create_time' : 1680013019
         })
```

Out[8]:

```
Document(page_content="This is my document. It is full of text that I've gathered from ot
her places", metadata={'my_document_id': 234234, 'my_document_source': 'The LangChain Pap
ers', 'my_document_create_time': 1680013019})
```

**But you don't have to include metadata if you don't want to**

In [9]:

```
Document(page_content="This is my document. It is full of text that I've gathered from ot
her places")
```

Out[9]:

```
Document(page_content="This is my document. It is full of text that I've gathered from ot
her places")
```

## Models - The interface to the AI brains

### Language Model

**A model that does text in ➡ text out!**

***Check out how I changed the model I was using from the default one to ada-001 (a very cheap, low performing model). See more models here***

In [10]:

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

```
llm = OpenAI(model_name="text-ada-001", openai_api_key=openai_api_key)
```

In [11]:

```
llm("What day comes after Friday?")
```

Out[11]:

```
'\n\nSaturday'
```

## Chat Model

**A model that takes a series of messages and returns a message output**

In [12]:

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage, AIMessage

chat = ChatOpenAI(temperature=1, openai_api_key=openai_api_key)
```

In [13]:

```
chat(
    [
        SystemMessage(content="You are an unhelpful AI bot that makes a joke at whatever 
the user says"),
        HumanMessage(content="I would like to go to New York, how should I do this?")
    ]
)
```

Out[13]:

```
AIMessage(content='Why did the math book go to New York? Because it had too many problems 
and needed a change of scenery!')
```

## Function Calling Models

**Function calling models are similar to Chat Models but with a little extra flavor. They are fine tuned to give structured data outputs.**

**This comes in handy when you're making an API call to an external service or doing extraction.**

In [14]:

```
chat = ChatOpenAI(model='gpt-3.5-turbo-0613', temperature=1, openai_api_key=openai_api_ke
y)

output = chat(messages=
     [
         SystemMessage(content="You are an helpful AI bot"),
         HumanMessage(content="What’s the weather like in Boston right now?")
     ],
     functions=[{
         "name": "get_current_weather",
         "description": "Get the current weather in a given location",
         "parameters": {
             "type": "object",
             "properties": {
                 "location": {
                     "type": "string",
                     "description": "The city and state, e.g. San Francisco, CA"
                 },
                 "unit": {
                     "type": "string",
                     "enum": ["celsius", "fahrenheit"]
                 }
             },
             "required": ["location"]
         }
     }
     ]
)
output
```

Out[14]:

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'get_current_weather',
'arguments': '{\n  "location": "Boston, MA"\n}'}})
```

**See the extra `additional_kwargs` that is passed back to us? We can take that and pass it to an external API to get data. It saves the hassle of doing output parsing.**

### Text Embedding Model

**Change your text into a vector (a series of numbers that hold the semantic 'meaning' of your text). Mainly used when comparing two pieces of text together.**

***BTW: Semantic means 'relating to meaning in language or logic.'***

In [15]:

```
from langchain.embeddings import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)
```

In [16]:

```
text = "Hi! It's time for the beach"
```

In [17]:

```
text_embedding = embeddings.embed_query(text)
print (f"Here's a sample: {text_embedding[:5]}...")
print (f"Your embedding is length {len(text_embedding)}")
```

```
Here's a sample: [-0.00019600906371495047, -0.003184673442291136З, -0.0007734206914647714
, -0.019472001962491232, -0.015092319017854244]...
Your embedding is length 1536
```

## Prompts - Text generally used as instructions to your model

### Prompt

**What you'll pass to the underlying model**

In [18]:

```
from langchain.llms import OpenAI

llm = OpenAI(model_name="text-davinci-003", openai_api_key=openai_api_key)

# I like to use three double quotation marks for my prompts because it's easier to read
prompt = """
Today is Monday, tomorrow is Wednesday.

What is wrong with that statement?
"""

print(llm(prompt))
```

```
The statement is incorrect. Tomorrow is Tuesday, not Wednesday.
```

## Prompt Template

**An object that helps create prompts based on a combination of user input, other non-static information and a fixed template string.**

**Think of it as an f-string in python but for prompts**

***Advanced: Check out LangSmithHub(https://smith.langchain.com/hub) for many more communit prompt templates***

In [19]:

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

llm = OpenAI(model_name="text-davinci-003", openai_api_key=openai_api_key)

# Notice "location" below, that is a placeholder for another value later
template = """
I really want to travel to {location}. What should I do there?

Respond in one short sentence
"""

prompt = PromptTemplate(
    input_variables=["location"],
    template=template,
)

final_prompt = prompt.format(location='Rome')

print (f"Final Prompt: {final_prompt}")
print ("-----------")
print (f"LLM Output: {llm(final_prompt)}")
```

```
Final Prompt:
I really want to travel to Rome. What should I do there?

Respond in one short sentence

-----------
LLM Output: Visit the Colosseum, the Vatican, and the Trevi Fountain.
```

## Example Selectors

**An easy way to select from a series of examples that allow you to dynamic place in-context information into your prompt. Often used when your task is nuanced or you have a large list of examples.**

**Check out different types of example selectors here**

**If you want an overview on why examples are important (prompt engineering), check out this video**

In [20]:

```
from langchain.prompts.example_selector import SemanticSimilarityExampleSelector
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain.prompts import FewShotPromptTemplate, PromptTemplate
from langchain.llms import OpenAI

llm = OpenAI(model_name="text-davinci-003", openai_api_key=openai_api_key)

example_prompt = PromptTemplate(
    input_variables=["input", "output"],
    template="Example Input: {input}\nExample Output: {output}",
)

# Examples of locations that nouns are found
examples = [
    {"input": "pirate", "output": "ship"},
    {"input": "pilot", "output": "plane"},
    {"input": "driver", "output": "car"},
    {"input": "tree", "output": "ground"},
    {"input": "bird", "output": "nest"},
]
```

```
/Users/gregorykamradt/opt/anaconda3/lib/python3.9/site-packages/deeplake/util/check_lates
t_version.py:32: UserWarning: A newer version of deeplake (3.7.2) is available. It's reco
mmended that you update to the latest version using `pip install -U deeplake`.
  warnings.warn(
```

In [21]:

```
# SemanticSimilarityExampleSelector will select examples that are similar to your input b
y semantic meaning

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 se
mantic similarity.
    OpenAIEmbeddings(openai_api_key=openai_api_key),

    # This is the VectorStore class that is used to store the embeddings and do a similar
ity search over.
    Chroma,

    # This is the number of examples to produce.
    k=2
)
```

In [22]:

```
similar_prompt = FewShotPromptTemplate(
    # The object that will help select examples
    example_selector=example_selector,

    # Your prompt
    example_prompt=example_prompt,

    # Customizations that will be added to the top and bottom of your prompt
    prefix="Give the location an item is usually found in",
    suffix="Input: {noun}\nOutput:",

    # What inputs your prompt will receive
    input_variables=["noun"],
)
```

In [23]:

```
# Select a noun!
my_noun = "plant"
# my_noun = "student"

print(similar_prompt.format(noun=my_noun))
```

```
Give the location an item is usually found in

Example Input: tree
Example Output: ground

Example Input: bird
Example Output: nest

Input: plant
Output:
```

In [24]:

```
llm(similar_prompt.format(noun=my_noun))
```

Out[24]:

```
' pot'
```

## Output Parsers Method 1: Prompt Instructions & String Parsing

**A helpful way to format the output of a model. Usually used for structured output. LangChain has a bunch more output parsers listed on their documentation.**

**Two big concepts:**

**1. Format Instructions - A autogenerated prompt that tells the LLM how to format it's response based off your desired result**

**2. Parser - A method which will extract your model's text output into a desired structure (usually json)**

In [25]:

```
from langchain.output_parsers import StructuredOutputParser, ResponseSchema
from langchain.prompts import ChatPromptTemplate, HumanMessagePromptTemplate
from langchain.llms import OpenAI
```

In [26]:

```
llm = OpenAI(model_name="text-davinci-003", openai_api_key=openai_api_key)
```

In [27]:

```
# How you would like your response structured. This is basically a fancy prompt template
response_schemas = [
    ResponseSchema(name="bad_string", description="This a poorly formatted user input st
ring"),
    ResponseSchema(name="good_string", description="This is your response, a reformatted
response")
]

# How you would like to parse your output
output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
```

In [28]:

```
# See the prompt template you created for formatting
format_instructions = output_parser.get_format_instructions()
print (format_instructions)
```

~~~
The output should be a markdown code snippet formatted in the following schema, including
the leading and trailing "```json" and "```":

```json
{
 "bad_string": string  // This a poorly formatted user input string
 "good_string": string  // This is your response, a reformatted response
}
```
~~~

In [29]:

```
template = """
You will be given a poorly formatted string from a user.
Reformat it and make sure all the words are spelled correctly

{format_instructions}

% USER INPUT:
{user_input}

YOUR RESPONSE:
"""
```

```
prompt = PromptTemplate(
    input_variables=["user_input"],
    partial_variables={"format_instructions": format_instructions},
    template=template
)

promptValue = prompt.format(user_input="welcom to califonya!")

print(promptValue)
```

~~~
You will be given a poorly formatted string from a user.
Reformat it and make sure all the words are spelled correctly

The output should be a markdown code snippet formatted in the following schema, including
the leading and trailing "```json" and "```":

```json
{
 "bad_string": string  // This a poorly formatted user input string
 "good_string": string  // This is your response, a reformatted response
}
```

% USER INPUT:
welcom to califonya!

YOUR RESPONSE:
~~~

In [30]:

```
llm_output = llm(promptValue)
llm_output
```

Out[30]:

```
'```json\n{\n\t"bad_string": "welcom to califonya!", \n\t"good_string": "Welcome to Calif
ornia!"\n}\n```'
```

In [31]:

```
output_parser.parse(llm_output)
```

Out[31]:

```
{'bad_string': 'welcom to califonya!', 'good_string': 'Welcome to California!'}
```

## Output Parsers Method 2: OpenAI Fuctions

**When OpenAI released function calling, the game changed. This is recommended method when starting out.**

**They trained models specifically for outputing structured data. It became super easy to specify a Pydantic schema and get a structured output.**

**There are many ways to define your schema, I prefer using Pydantic Models because of how organized they are. Feel free to reference OpenAI's documention for other methods.**

**In order to use this method you'll need to use a model that supports function calling. I'll use** `gpt4-0613`

**Example 1: Simple**

**Let's get started by defining a simple model for us to extract from.**

In [32]:

```
from langchain.pydantic_v1 import BaseModel, Field
from typing import Optional

class Person(BaseModel):
    """Identifying information about a person."""
```

```
    name: str = Field(..., description="The person's name")
    age: int = Field(..., description="The person's age")
    fav_food: Optional[str] = Field(None, description="The person's favorite food")
```

**Then let's create a chain (more on this later) that will do the extracting for us**

In [33]:

```
from langchain.chains.openai_functions import create_structured_output_chain

llm = ChatOpenAI(model='gpt-4-0613', openai_api_key=openai_api_key)

chain = create_structured_output_chain(Person, llm, prompt)
chain.run(
    "Sally is 13, Joey just turned 12 and loves spinach. Caroline is 10 years older than 
Sally."
)
```

Out[33]:

```
Person(name='Sally, Joey, Caroline', age=13, fav_food='spinach')
```

**Notice how we only have data on one person from that list? That is because we didn't specify we wanted multiple. Let's change our schema to specify that we want a list of people if possible.**

In [34]:

```
from typing import Sequence

class People(BaseModel):
    """Identifying information about all people in a text."""

    people: Sequence[Person] = Field(..., description="The people in the text")
```

**Now we'll call for People rather than Person**

In [35]:

```
chain = create_structured_output_chain(People, llm, prompt)
chain.run(
    "Sally is 13, Joey just turned 12 and loves spinach. Caroline is 10 years older than 
Sally."
)
```

Out[35]:

```
People(people=[Person(name='Sally', age=13, fav_food=None), Person(name='Joey', age=12, f
av_food='spinach'), Person(name='Caroline', age=23, fav_food=None)])
```

**Let's do some more parsing with it**

**Example 2: Enum**

**Now let's parse when a product from a list is mentioned**

In [36]:

```
import enum

llm = ChatOpenAI(model='gpt-4-0613', openai_api_key=openai_api_key)

class Product(str, enum.Enum):
    CRM = "CRM"
    VIDEO_EDITING = "VIDEO_EDITING"
    HARDWARE = "HARDWARE"
```

In [37]:

```
class Products(BaseModel):
    """Identifying products that were mentioned in a text"""

    products: Sequence[Product] = Field(..., description="The products mentioned in a tex
t")
```

In [38]:

```
chain = create_structured_output_chain(Products, llm, prompt)
chain.run(
    "The CRM in this demo is great. Love the hardware. The microphone is also cool. Love
the video editing"
)
```

Out[38]:

```
Products(products=[<Product.CRM: 'CRM'>, <Product.HARDWARE: 'HARDWARE'>, <Product.VIDEO_E
DITING: 'VIDEO_EDITING'>])
```

## Indexes - Structuring documents to LLMs can work with them

### Document Loaders

**Easy ways to import data from other sources. Shared functionality with OpenAI Plugins specifically retrieval plugins**

**See a big list of document loaders here. A bunch more on Llama Index as well.**

#### HackerNews

In [39]:

```
from langchain.document_loaders import HNLoader
```

In [40]:

```
loader = HNLoader("https://news.ycombinator.com/item?id=34422627")
```

In [41]:

```
data = loader.load()
```

In [42]:

```
print (f"Found {len(data)} comments")
print (f"Here's a sample:\n\n{''.join([x.page_content[:150] for x in data[:2]])}")
```

```
Found 76 comments
Here's a sample:

Ozzie_osman 8 months ago
             | next [-]

LangChain is awesome. For people not sure what it's doing, large language models (LLMs) a
re very Ozzie_osman 8 months ago
             | parent | next [-]

Also, another library to check out is GPT Index (https://github.com/jerryjliu/gpt_index)
```

#### Books from Gutenberg Project

In [43]:

```
from langchain.document_loaders import GutenbergLoader

loader = GutenbergLoader("https://www.gutenberg.org/cache/epub/2148/pg2148.txt")
```

```
data = loader.load()
```

In [44]:

```
print(data[0].page_content[1855:1984])
```

```
      At Paris, just after dark one gusty evening in the autumn of 18-,


      I was enjoying the twofold luxury of meditation
```

**URLs and webpages**

**Let's try it out with [Paul Graham's website]**

In [45]:

```
from langchain.document_loaders import UnstructuredURLLoader

urls = [
    "http://www.paulgraham.com/",
]

loader = UnstructuredURLLoader(urls=urls)

data = loader.load()

data[0].page_content
```

Out[45]:

```
'New: \n\nHow to Do Great Work |\nRead |\nWill |\nTruth\n\n\n\n\n\nWant to start a startu
p? Get funded by Y Combinator.\n\n\n\n\n\n\n\n\n\n© mmxxiii pg'
```

## Text Splitters

**Often times your document is too long (like a book) for your LLM. You need to split it up into chunks. Text splitters help with this.**

**There are many ways you could split your text into chunks, experiment with [different ones] to see which is best for you.**

In [46]:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

In [47]:

```
# This is a long document we can split up.
with open('data/PaulGrahamEssays/worked.txt') as f:
    pg_work = f.read()

print (f"You have {len([pg_work])} document")
```

```
You have 1 document
```

In [48]:

```
text_splitter = RecursiveCharacterTextSplitter(
    # Set a really small chunk size, just to show.
    chunk_size = 150,
    chunk_overlap  = 20,
)

texts = text_splitter.create_documents([pg_work])
```

In [49]:

```
print (f"You have {len(texts)} documents")
```

```
You have 610 documents
```

In [50]:

```python
print ("Preview:")
print (texts[0].page_content, "\n")
print (texts[1].page_content)
```

```
Preview:
February 2021Before college the two main things I worked on, outside of school,
were writing and programming. I didn't write essays. I wrote what

beginning writers were supposed to write then, and probably still
are: short stories. My stories were awful. They had hardly any plot,
```

**There are a ton of different ways to do text splitting and it really depends on your retrieval strategy and application design. Check out more splitters here**

## Retrievers

**Easy way to combine documents with language models.**

**There are many different types of retrievers, the most widely supported is the VectoreStoreRetriever**

In [51]:

```python
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

loader = TextLoader('data/PaulGrahamEssays/worked.txt')
documents = loader.load()
```

In [52]:

```python
# Get your splitter ready
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=50)

# Split your docs into texts
texts = text_splitter.split_documents(documents)

# Get embedding engine ready
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)

# Embedd your texts
db = FAISS.from_documents(texts, embeddings)
```

In [53]:

```python
# Init your retriever. Asking for just 1 document back
retriever = db.as_retriever()
```

In [54]:

```python
retriever
```

Out[54]:

```
VectorStoreRetriever(tags=['FAISS'], vectorstore=<langchain.vectorstores.faiss.FAISS obje
ct at 0x7f8389169070>)
```

In [55]:

```python
docs = retriever.get_relevant_documents("what types of things did the author want to buil
d?")
```

In [56]:

```
print("\n\n".join([x.page_content[:200] for x in docs[:2]]))
```

```
standards; what was the point? No one else wanted one either, so
off they went. That was what happened to systems work.I wanted not just to build things,
but to build things that would
last.In this di

much of it in grad school.Computer Science is an uneasy alliance between two halves, theo
ry
and systems. The theory people prove things, and the systems people
build things. I wanted to build things.
```

## VectorStores

**Databases to store vectors. Most popular ones are [Pinecone](#) & [Weaviate](#). More examples on OpenAIs [retriever documentation](#). [Chroma](#) & [FAISS](#) are easy to work with locally.**

**Conceptually, think of them as tables w/ a column for embeddings (vectors) and a column for metadata.**

**Example**

| Embedding | Metadata |
|---|---|
| [-0.00015641732898075134, -0.003165106289088726, ...] | {'date' : '1/2/23} |
| [-0.00035465431654651654, 1.4654131651654516546, ...] | {'date' : '1/3/23} |

In [57]:

```
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

loader = TextLoader('data/PaulGrahamEssays/worked.txt')
documents = loader.load()

# Get your splitter ready
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=50)

# Split your docs into texts
texts = text_splitter.split_documents(documents)

# Get embedding engine ready
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)
```

In [58]:

```
print (f"You have {len(texts)} documents")
```

```
You have 78 documents
```

In [59]:

```
embedding_list = embeddings.embed_documents([text.page_content for text in texts])
```

In [60]:

```
print (f"You have {len(embedding_list)} embeddings")
print (f"Here's a sample of one: {embedding_list[0][:3]}...")
```

```
You have 78 embeddings
Here's a sample of one: [-0.001058628615053026, -0.01118234211553424, -0.0128748047462668
83]...
```

**Your vectorstore store your embeddings (☝️) and make them easily searchable**

## Memory

**Helping LLMs remember information.**

**Memory is a bit of a loose term. It could be as simple as remembering information you've chatted about in the past or more complicated information retrieval.**

**We'll keep it towards the Chat Message use case. This would be used for chat bots.**

**There are many types of memory, explore [the documentation] to see which one fits your use case.**

### Chat Message History

In [61]:

```
from langchain.memory import ChatMessageHistory
from langchain.chat_models import ChatOpenAI

chat = ChatOpenAI(temperature=0, openai_api_key=openai_api_key)

history = ChatMessageHistory()

history.add_ai_message("hi!")

history.add_user_message("what is the capital of france?")
```

In [62]:

```
history.messages
```

Out[62]:

```
[AIMessage(content='hi!'),
 HumanMessage(content='what is the capital of france?')]
```

In [63]:

```
ai_response = chat(history.messages)
ai_response
```

Out[63]:

```
AIMessage(content='The capital of France is Paris.')
```

In [64]:

```
history.add_ai_message(ai_response.content)
history.messages
```

Out[64]:

```
[AIMessage(content='hi!'),
 HumanMessage(content='what is the capital of france?'),
 AIMessage(content='The capital of France is Paris.')]
```

## Chains ▯▯▯

**Combining different LLM calls and action automatically**

**Ex: Summary #1, Summary #2, Summary #3 > Final Summary**

**Check out this video explaining different summarization chain types**

**There are many applications of chains search to see which are best for your use case.**

**We'll cover two of them:**

### 1. Simple Sequential Chains

**Easy chains where you can use the output of an LLM as an input into another. Good for breaking up tasks (and**

**keeping your LLM focused)**

In [65]:

```
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.chains import SimpleSequentialChain

llm = OpenAI(temperature=1, openai_api_key=openai_api_key)
```

In [66]:

```
template = """Your job is to come up with a classic dish from the area that the users sug
gests.
% USER LOCATION
{user_location}

YOUR RESPONSE:
"""
prompt_template = PromptTemplate(input_variables=["user_location"], template=template)

# Holds my 'location' chain
location_chain = LLMChain(llm=llm, prompt=prompt_template)
```

In [67]:

```
template = """Given a meal, give a short and simple recipe on how to make that dish at ho
me.
% MEAL
{user_meal}

YOUR RESPONSE:
"""
prompt_template = PromptTemplate(input_variables=["user_meal"], template=template)

# Holds my 'meal' chain
meal_chain = LLMChain(llm=llm, prompt=prompt_template)
```

In [68]:

```
overall_chain = SimpleSequentialChain(chains=[location_chain, meal_chain], verbose=True)
```

In [69]:

```
review = overall_chain.run("Rome")
```

```
> Entering new SimpleSequentialChain chain...

A classic dish from Rome is Spaghetti alla Carbonara, featuring egg, Parmesan cheese, bla
ck pepper, and pancetta or guanciale.

Ingredients:
- 8oz spaghetti
- 4 tablespoons olive oil
- 4oz diced pancetta or guanciale
- 2 cloves garlic, minced
- 2 eggs, lightly beaten
- 2 tablespoons parsley, chopped
- ½ cup grated Parmesan
- Salt and black pepper to taste

Instructions:
1. Bring a pot of salted water to a boil and add the spaghetti. Cook according to package
directions.
2. Meanwhile, add the olive oil to a large skillet over medium-high heat. Add the diced p
ancetta and garlic, and cook until pancetta is browned and garlic is fragrant.
3. In a medium bowl, whisk together the eggs, parsley, Parmesan, and salt and pepper.
4. Drain the cooked spaghetti and add it to the skillet with the pancetta and garlic. Rem
ove from heat and pour the egg mixture over the spaghetti, stirring to combine.
```

```
5. Serve the spaghetti alla carbonara with additional Parmesan cheese and black pepper.

> Finished chain.
```

## 2. Summarization Chain

**Easily run through long numerous documents and get a summary. Check out this video for other chain types besides map-reduce**

In [70]:

```
from langchain.chains.summarize import load_summarize_chain
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

loader = TextLoader('data/PaulGrahamEssays/disc.txt')
documents = loader.load()

# Get your splitter ready
text_splitter = RecursiveCharacterTextSplitter(chunk_size=700, chunk_overlap=50)

# Split your docs into texts
texts = text_splitter.split_documents(documents)

# There is a lot of complexity hidden in this one line. I encourage you to check out the
video above for more detail
chain = load_summarize_chain(llm, chain_type="map_reduce", verbose=True)
chain.run(texts)
```

```
> Entering new MapReduceDocumentsChain chain...


> Entering new LLMChain chain...
Prompt after formatting:
Write a concise summary of the following:


"January 2017Because biographies of famous scientists tend to
edit out their mistakes, we underestimate the
degree of risk they were willing to take.
And because anything a famous scientist did that
wasn't a mistake has probably now become the
conventional wisdom, those choices don't
seem risky either.Biographies of Newton, for example, understandably focus
more on physics than alchemy or theology.
The impression we get is that his unerring judgment
led him straight to truths no one else had noticed.
How to explain all the time he spent on alchemy
and theology?  Well, smart people are often kind of
crazy.But maybe there is a simpler explanation. Maybe"


CONCISE SUMMARY:
Prompt after formatting:
Write a concise summary of the following:


"the smartness and the craziness were not as separate
as we think. Physics seems to us a promising thing
to work on, and alchemy and theology obvious wastes
of time. But that's because we know how things
turned out. In Newton's day the three problems
seemed roughly equally promising. No one knew yet
what the payoff would be for inventing what we
now call physics; if they had, more people would
have been working on it. And alchemy and theology
were still then in the category Marc Andreessen would
describe as "huge, if true."Newton made three bets. One of them worked. But
they were all risky."
```

```
CONCISE SUMMARY:

> Finished chain.


> Entering new LLMChain chain...
Prompt after formatting:
Write a concise summary of the following:


" Biographies of famous scientists often edit out their mistakes, giving readers the wron
g impression that they never faced any risks to achieve successful results. An example of
this is Newton, whose smartness is assumed to have led straight him to truths without any
detours into alchemy or theology - despite the fact that he spent a lot of time on both f
ields. Maybe the simpler explanation is that he was willing to take risks, even if it mea
ns potentially making mistakes.

 In the 17th century, Newton took a risk and made three bets, one of which turned out to
be a successful invention of what we now call physics. The other two bets were on less po
pular subjects of the time such as alchemy and theology. People did not know then what th
e payoff would be, but the bets still seemed relatively promising."


CONCISE SUMMARY:

> Finished chain.

> Finished chain.
```

```
Out[70]:
```

```
" Biographies tend to omit famous scientists' mistakes from their stories, but Newton was
willing to take risks and explore multiple fields to make his discoveries. He placed thre
e risky bets, one of which resulted in the creation of physics as we know it today."
```

## Agents □□

Official LangChain Documentation describes agents perfectly (emphasis mine):

> Some applications will require not just a predetermined chain of calls to LLMs/other tools, but potentially an **unknown chain** that depends on the user's input. In these types of chains, there is a “agent” which has access to a suite of tools. Depending on the user input, the agent can then **decide which, if any, of these tools to call** .

Basically you use the LLM not just for text output, but also for decision making. The coolness and power of this functionality can't be overstated enough.

Sam Altman emphasizes that the LLMs are good ' reasoning engine'. Agent take advantage of this.

### Agents

The language model that drives decision making.

More specifically, an agent takes in an input and returns a response corresponding to an action to take along with an action input. You can see different types of agents (which are better for different use cases) here.

### Tools

A 'capability' of an agent. This is an abstraction on top of a function that makes it easy for LLMs (and agents) to interact with it. Ex: Google search.

This area shares commonalities with OpenAI plugins.

## Toolkit

**Groups of tools that your agent can select from**

**Let's bring them all together:**

In [71]:

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

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

In [72]:

```
serpapi_api_key=os.getenv("SERP_API_KEY", "YourAPIKey")
```

In [73]:

```
toolkit = load_tools(["serpapi"], llm=llm, serpapi_api_key=serpapi_api_key)
```

In [74]:

```
agent = initialize_agent(toolkit, llm, agent="zero-shot-react-description", verbose=True
, return_intermediate_steps=True)
```

In [75]:

```
response = agent({"input":"what was the first album of the"
                  "band that Natalie Bergman is a part of?"})
```

```
> Entering new AgentExecutor chain...
 I should try to find out what band Natalie Bergman is a part of.
Action: Search
Action Input: "Natalie Bergman band"
Observation: ['Natalie Bergman is an American singer-songwriter. She is one half of the d
uo Wild Belle, along with her brother Elliot Bergman. Her debut solo album, Mercy, was re
leased on Third Man Records on May 7, 2021. She is based in Los Angeles.', 'Natalie Bergm
an type: American singer-songwriter.', 'Natalie Bergman main_tab_text: Overview.', 'Natal
ie Bergman kgmid: /m/0qgx4kh.', 'Natalie Bergman genre: Folk.', 'Natalie Bergman parents:
Susan Bergman, Judson Bergman.', 'Natalie Bergman born: 1988 or 1989 (age 34-35).', 'Nata
lie Bergman is an American singer-songwriter. She is one half of the duo Wild Belle, alon
g with her brother Elliot Bergman. Her debut solo album, Mercy, ...']
Thought: I should search for the first album of Wild Belle
Action: Search
Action Input: "Wild Belle first album"
Observation: Isles
Thought: I now know the final answer
Final Answer: Isles is the first album of the band that Natalie Bergman is a part of.

> Finished chain.
```

**Enjoy** https://open.spotify.com/track/1eREJIBdqeCcqNCB1pbz7w?si=c014293b63c7478c

# LangChain Cookbook Part 2: Use Cases□□□□

*This cookbook is based on the LangChain Conceptual Documentation*

**Goals:**

1. Inspire you to build
2. Provide an introductory understanding of the main use cases of LangChain via ELI5 examples and code snippets. For an introduction to the *fundamentals* of LangChain check out Cookbook Part 1: Fundamentals.

**LangChain Links:**

- LC Conceptual Documentation
- LC Python Documentation
- LC Javascript/Typescript Documentation
- LC Discord
- www.langchain.com
- LC Twitter

## What is LangChain?

> LangChain is a framework for developing applications powered by language models. *Source*

**TLDR**: LangChain makes the complicated parts of working & building with AI models easier. It helps do this in two ways:

1. **Integration** - Bring external data, such as your files, other applications, and api data, to your LLMs
2. **Agency** - Allow your LLMs to interact with its environment via decision making. Use LLMs to help decide which action to take next

## Why LangChain?

1. **Components** - LangChain makes it easy to swap out abstractions and components necessary to work with language models.
2. **Customized Chains** - LangChain provides out of the box support for using and customizing 'chains' - a series of actions strung together.
3. **Speed** □ - This team ships insanely fast. You'll be up to date with the latest LLM features.
4. **Community** □ - Wonderful discord and community support, meet ups, hackathons, etc.

Though LLMs can be straightforward (text-in, text-out) you'll quickly run into friction points that LangChain helps with once you develop more complicated applications.

## Main Use Cases

- **Summarization** - Express the most important facts about a body of text or chat interaction
- **Question and Answering Over Documents** - Use information held within documents to answer questions or query
- **Extraction** - Pull structured data from a body of text or an user query
- **Evaluation** - Understand the quality of output from your application
- **Querying Tabular Data** - Pull data from databases or other tabular source
- **Code Understanding** - Reason about and digest code
- **Interacting with APIs** - Query APIs and interact with the outside world
- **Chatbots** - A framework to have a back and forth interaction with a user combined with memory in a chat interface
- **Agents** - Use LLMs to make decisions about what to do next. Enable these decisions with tools.

- **Agents** - Use LLMs to make decisions about what to do next. Enable these decisions with tools.

Want to see live examples of these use cases? Head over to the [LangChain Project Gallery]

**Authors Note:**

- This cookbook will not cover all aspects of LangChain. It's contents have been curated to get you to building & impact as quick as possible. For more, please check out [LangChain Technical Documentation]
- This notebook assumes is that you've seen part 1 of this series [Fundamentals]. This notebook is focused on what to do and how to apply those fundamentals.
- You'll notice I repeat import statements throughout the notebook. My intention is to lean on the side of clarity and help you see the full code block in one spot. No need to go back and forth to see when we imported a package.
- We use the default models throughout the notebook, at the time of writing they were davinci-003 and gpt-3.5-turbo. You would no doubt get better results with GPT4

Let's get started

Throughout this tutorial we will use OpenAI's various [models]. LangChain makes it easy to [subsistute LLMs] so you can BYO-LLM if you want

In [1]:

```
from dotenv import load_dotenv
import os

load_dotenv()

openai_api_key = os.getenv('OPENAI_API_KEY', 'YourAPIKeyIfNotSet')
```

In [2]:

```
# Run this cell if you want to make your display wider
from IPython.display import display, HTML
display(HTML("<style>.container { width:90% !important; }</style>"))
```

# LangChain Use Cases

## Summarization

One of the most common use cases for LangChain and LLMs is summarization. You can summarize any piece of text, but use cases span from summarizing calls, articles, books, academic papers, legal documents, user history, a table, or financial documents. It's super helpful to have a tool which can summarize information quickly.

- **Deep Dive** - (Coming Soon)
- **Examples** - [Summarizing B2B Sales Calls]
- **Use Cases** - Summarize Articles, Transcripts, Chat History, Slack/Discord, Customer Interactions, Medical Papers, Legal Documents, Podcasts, Tweet Threads, Code Bases, Product Reviews, Financial Documents

### Summaries Of Short Text

For summaries of short texts, the method is straightforward, in fact you don't need to do anything fancy other than simple prompting with instructions

In [3]:

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

```
# Note, the default model is already 'text-davinci-003' but I call it out here
explicitly so you know where to change it later if you want
llm = OpenAI(temperature=0, model_name='text-davinci-003', openai_api_key=opena
i_api_key)

# Create our template
template = """
%INSTRUCTIONS:
Please summarize the following piece of text.
Respond in a manner that a 5 year old would understand.

%TEXT:
{text}
"""

# Create a LangChain prompt template that we can insert values to later
prompt = PromptTemplate(
    input_variables=["text"],
    template=template,
)
```

**Let's let's find a confusing text online.** ***Source***

In [4]:

```
confusing_text = """
For the next 130 years, debate raged.
Some scientists called Prototaxites a lichen, others a fungus, and still others
clung to the notion that it was some kind of tree.
“The problem is that when you look up close at the anatomy, it’s evocative of a
lot of different things, but it’s diagnostic of nothing,” says Boyce, an associ
ate professor in geophysical sciences and the Committee on Evolutionary Biology
.
“And it’s so damn big that when whenever someone says it’s something, everyone
else’s hackles get up: ‘How could you have a lichen 20 feet tall?’”
"""
```

**Let's take a look at what prompt will be sent to the LLM**

In [5]:

```
print ("------- Prompt Begin -------")

final_prompt = prompt.format(text=confusing_text)
print(final_prompt)

print ("------- Prompt End -------")
```

```
------- Prompt Begin -------

%INSTRUCTIONS:
Please summarize the following piece of text.
Respond in a manner that a 5 year old would understand.

%TEXT:

For the next 130 years, debate raged.
Some scientists called Prototaxites a lichen, others a fungus, and still others
clung to the notion that it was some kind of tree.
“The problem is that when you look up close at the anatomy, it’s evocative of a
lot of different things, but it’s diagnostic of nothing,” says Boyce, an associa
te professor in geophysical sciences and the Committee on Evolutionary Biology.
“And it’s so damn big that when whenever someone says it’s something, everyone e
lse’s hackles get up: ‘How could you have a lichen 20 feet tall?’”


------- Prompt End -------
```

**Finally let's pass it through the LLM**

In [6]:

```
output = llm(final_prompt)
print (output)
```

```
For 130 years, people argued about what Prototaxites was. Some thought it was a
lichen, some thought it was a fungus, and some thought it was a tree. But no one
could agree. It was so big that it was hard to figure out what it was.
```

**This method works fine, but for longer text, it can become a pain to manage and you'll run into token limits. Luckily LangChain has out of the box support for different methods to summarize via their load_summarize_chain.**

## Summaries Of Longer Text

***Note: This method will also work for short text too***

In [7]:

```
from langchain.llms import OpenAI
from langchain.chains.summarize import load_summarize_chain
from langchain.text_splitter import RecursiveCharacterTextSplitter

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

**Let's load up a longer document**

In [8]:

```
with open('data/PaulGrahamEssays/good.txt', 'r') as file:
    text = file.read()

# Printing the first 285 characters as a preview
print (text[:285])
```

```
April 2008(This essay is derived from a talk at the 2008 Startup School.)About a
month after we started Y Combinator we came up with the
phrase that became our motto: Make something people want.  We've
learned a lot since then, but if I were choosing now that's still
the one I'd pick.
```

**Then let's check how many tokens are in this document. get_num_tokens is a nice method for this.**

In [9]:

```
num_tokens = llm.get_num_tokens(text)

print (f"There are {num_tokens} tokens in your file")
```

```
There are 3970 tokens in your file
```

**While you could likely stuff this text in your prompt, let's act like it's too big and needs another method.**

**First we'll need to split it up. This process is called 'chunking' or 'splitting' your text into smaller pieces. I like the RecursiveCharacterTextSplitter because it's easy to control but there are a bunch you can try**

In [10]:

```
text_splitter = RecursiveCharacterTextSplitter(separators=["\n\n", "\n"], chunk
_size=5000, chunk_overlap=350)
docs = text_splitter.create_documents([text])

print (f"You now have {len(docs)} docs intead of 1 piece of text")
```

```
You now have 4 docs intead of 1 piece of text
```

**Next we need to load up a chain which will make successive calls to the LLM for us. Want to see the prompt being used in the chain below? Check out the LangChain documentation**

**For information on the difference between chain types, check out this video on token limit workarounds**

***Note: You could also get fancy and make the first 4 calls of the map_reduce run in parallel too***

In [11]:

```
# Get your chain ready to use
chain = load_summarize_chain(llm=llm, chain_type='map_reduce') # verbose=True o
ptional to see what is getting sent to the LLM
```

In [12]:

```
# Use it. This will run through the 4 documents, summarize the chunks, then get
a summary of the summary.
output = chain.run(docs)
print (output)
```

```
 This essay looks at the idea of benevolence in startups, and how it can help th
em succeed. It explains how benevolence can improve morale, make people want to
help, and help startups be decisive. It also looks at how markets have evolved t
o value potential dividends and potential earnings, and how users dislike their
new operating system. The author argues that starting a company with benevolent
aims is currently undervalued, and that Y Combinator's motto of "Make something
people want" is a powerful concept.
```

## Question & Answering Using Documents As Context

***LangChain Question & Answer Docs***

**In order to use LLMs for question and answer we must:**

1. **Pass the LLM relevant context it needs to answer a question**
2. **Pass it our question that we want answered**

**Simplified, this process looks like this "llm(your context + your question) = your answer"**

- **Deep Dive - Question A Book, Ask Questions To Your Custom Files, Chat Your Data JS (1000 pages of Financial Reports), LangChain Q&A webinar**
- **Examples - ChatPDF**
- **Use Cases - Chat your documents, ask questions to academic papers, create study guides, reference medical information**

### Simple Q&A Example

**Here let's review the convention of** `llm(your context + your question) = your answer`

In [13]:

```
from langchain.llms import OpenAI

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

In [14]:

```
context = """
Rachel is 30 years old
Bob is 45 years old
Kevin is 65 years old
"""
```

```
question = "Who is under 40 years old?"
```

**Then combine them.**

In [15]:

```
output = llm(context + question)

# I strip the text to remove the leading and trailing whitespace
print (output.strip())
```

```
Rachel is under 40 years old.
```

**As we ramp up our sophistication, we'll take advantage of this convention more.**

**The hard part comes in when you need to be selective about *which* data you put in your context. This field of study is called "document retrieval" and tightly coupled with AI Memory.**

## Using Embeddings

**I informally call what were about to go through as "The VectorStore Dance". It's the process of splitting your text, embedding the chunks, putting the embeddings in a DB, and then querying them. For a full video on this check out How To Question A Book**

**The goal is to select relevant chunks of our long text, but which chunks do we pull? The most popular method is to pull *similar* texts based off comparing vector embeddings.**

In [16]:

```
from langchain import OpenAI

# The vectorstore we'll be using
from langchain.vectorstores import FAISS

# The LangChain component we'll use to get the documents
from langchain.chains import RetrievalQA

# The easy document loader for text
from langchain.document_loaders import TextLoader

# The embedding engine that will convert our text to vectors
from langchain.embeddings.openai import OpenAIEmbeddings

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

**Let's load up a longer document**

In [17]:

```
loader = TextLoader('data/PaulGrahamEssays/worked.txt')
doc = loader.load()
print (f"You have {len(doc)} document")
print (f"You have {len(doc[0].page_content)} characters in that document")
```

```
You have 1 document
You have 74663 characters in that document
```

**Now let's split our long doc into smaller pieces**

In [18]:

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=3000, chunk_overlap=4
00)
docs = text_splitter.split_documents(doc)
```

In [19]:

```
# Get the total number of characters so we can see the average later
num_total_characters = sum([len(x.page_content) for x in docs])

print (f"Now you have {len(docs)} documents that have an average of {num_total_
characters / len(docs):,.0f} characters (smaller pieces)")
```

```
Now you have 29 documents that have an average of 2,930 characters (smaller piec
es)
```

In [20]:

```
# Get your embeddings engine ready
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)

# Embed your documents and combine with the raw text in a pseudo db. Note: This
will make an API call to OpenAI
docsearch = FAISS.from_documents(docs, embeddings)
```

**Create your retrieval engine**

In [21]:

```
qa = RetrievalQA.from_chain_type(llm=llm, chain_type="stuff", retriever=docsear
ch.as_retriever())
```

**Now it's time to ask a question. The retriever will go get the similar documents and combine with your question for the LLM to reason through.**

**Note: It may not seem like much, but the magic here is that we didn't have to pass in our full original document.**

In [22]:

```
query = "What does the author describe as good work?"
qa.run(query)
```

Out[22]:

```
' The author describes painting as good work.'
```

**If you wanted to do more you would hook this up to a cloud vector database, use a tool like metal and start managing your documents, with external data sources**

## Extraction

***LangChain Extraction Docs***

**Extraction is the process of parsing data from a piece of text. This is commonly used with output parsing in order to *structure* our data.**

- **Deep Dive - Use LLMs to Extract Data From Text (Expert Level Text Extraction , Structured Output From OpenAI (Clean Dirty Data)**
- **Examples - OpeningAttributes**
- **Use Cases: Extract a structured row from a sentence to insert into a database, extract multiple rows from a long document to insert into a database, extracting parameters from a user query to make an API call**

**A popular library for extraction is Kor. We won't cover it today but I highly suggest checking it out for advanced extraction.**

In [23]:

```
# To help construct our Chat Messages
from langchain.schema import HumanMessage
from langchain.prompts import PromptTemplate, ChatPromptTemplate, HumanMessageP
```

```
romptTemplate

# We will be using a chat model, defaults to gpt-3.5-turbo
from langchain.chat_models import ChatOpenAI

# To parse outputs and get structured data back
from langchain.output_parsers import StructuredOutputParser, ResponseSchema

chat_model = ChatOpenAI(temperature=0, model_name='gpt-3.5-turbo', openai_api_k
ey=openai_api_key)
```

## Vanilla Extraction

**Let's start off with an easy example. Here I simply supply a prompt with instructions with the type of output I want.**

In [24]:

```
instructions = """
You will be given a sentence with fruit names, extract those fruit names and as
sign an emoji to them
Return the fruit name and emojis in a python dictionary
"""

fruit_names = """
Apple, Pear, this is an kiwi
"""
```

In [25]:

```
# Make your prompt which combines the instructions w/ the fruit names
prompt = (instructions + fruit_names)

# Call the LLM
output = chat_model([HumanMessage(content=prompt)])

print (output.content)
print (type(output.content))
```

```
{'Apple': '□', 'Pear': '□', 'kiwi': '□'}
<class 'str'>
```

**Let's turn this into a proper python dictionary**

In [26]:

```
output_dict = eval(output.content)

print (output_dict)
print (type(output_dict))
```

```
{'Apple': '□', 'Pear': '□', 'kiwi': '□'}
<class 'dict'>
```

**While this worked this time, it's not a long term reliable method for more advanced use cases**

## Using LangChain's Response Schema

**LangChain's response schema will does two things for us:**

1. **Autogenerate the a prompt with bonafide format instructions. This is great because I don't need to worry about the prompt engineering side, I'll leave that up to LangChain!**
2. **Read the output from the LLM and turn it into a proper python object for me**

**Here I define the schema I want. I'm going to pull out the song and artist that a user wants to play from a pseudo chat message.**

In [27]:

```
# The schema I want out
response_schemas = [
    ResponseSchema(name="artist", description="The name of the musical artist")
,
    ResponseSchema(name="song", description="The name of the song that the arti
st plays")
]

# The parser that will look for the LLM output in my schema and return it back
to me
output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
```

In [28]:

```
# The format instructions that LangChain makes. Let's look at them
format_instructions = output_parser.get_format_instructions()
print(format_instructions)
```

~~~
The output should be a markdown code snippet formatted in the following schema,
including the leading and trailing "\`\`\`json" and "\`\`\`":

```json
{
 "artist": string  // The name of the musical artist
 "song": string  // The name of the song that the artist plays
}
~~~
```

In [29]:

```
# The prompt template that brings it all together
# Note: This is a different prompt template than before because we are using a
Chat Model

prompt = ChatPromptTemplate(
    messages=[
        HumanMessagePromptTemplate.from_template("Given a command from the user
, extract the artist and song names \n \
                                                    {format_instructions}\n{us
er_prompt}")
    ],
    input_variables=["user_prompt"],
    partial_variables={"format_instructions": format_instructions}
)
```

In [30]:

```
fruit_query = prompt.format_prompt(user_prompt="I really like So Young by Portu
gal. The Man")
print (fruit_query.messages[0].content)
```

```
Given a command from the user, extract the artist and song names
                                                    The output should be a ma
rkdown code snippet formatted in the following schema, including the leading and
trailing "\`\`\`json" and "\`\`\`":

```json
{
 "artist": string  // The name of the musical artist
 "song": string  // The name of the song that the artist plays
}
```
I really like So Young by Portugal. The Man
```

In [31]:

```
fruit_output = chat_model(fruit_query.to_messages())
output = output_parser.parse(fruit_output.content)
```
```

```
print (output)
print (type(output))
```

```
{'artist': 'Portugal. The Man', 'song': 'So Young'}
<class 'dict'>
```

**Awesome, now we have a dictionary that we can use later down the line**

**Warning: The parser looks for an output from the LLM in a specific format. Your model may not output the same format every time. Make sure to handle errors with this one. GPT4 and future iterations will be more reliable.**

**For more advanced parsing check out Kor**

## Evaluation

***LangChain Evaluation Docs***

**Evaluation is the process of doing quality checks on the output of your applications. Normal, deterministic, code has tests we can run, but judging the output of LLMs is more difficult because of the unpredictableness and variability of natural language. LangChain provides tools that aid us in this journey.**

- **Deep Dive - Coming Soon**
- **Examples - Lance Martin's Advanced Auto-Evaluator**
- **Use Cases: Run quality checks on your summarization or Question & Answer pipelines, check the output of you summarization pipeline**

In [32]:

```
# Embeddings, store, and retrieval
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS
from langchain.chains import RetrievalQA

# Model and doc loader
from langchain import OpenAI
from langchain.document_loaders import TextLoader

# Eval!
from langchain.evaluation.qa import QAEvalChain

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

In [33]:

```
# Our long essay from before
loader = TextLoader('data/PaulGrahamEssays/worked.txt')
doc = loader.load()

print (f"You have {len(doc)} document")
print (f"You have {len(doc[0].page_content)} characters in that document")
```

```
You have 1 document
You have 74663 characters in that document
```

**First let's do the Vectorestore dance so we can do question and answers**

In [34]:

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=3000, chunk_overlap=4
00)
docs = text_splitter.split_documents(doc)

# Get the total number of characters so we can see the average later
num_total_characters = sum([len(x.page_content) for x in docs])
```

```
print (f"Now you have {len(docs)} documents that have an average of {num_total_
characters / len(docs):,.0f} characters (smaller pieces)")
```

```
Now you have 29 documents that have an average of 2,930 characters (smaller piec
es)
```

In [35]:

```
# Embeddings and docstore
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)
docsearch = FAISS.from_documents(docs, embeddings)
```

**Make your retrieval chain. Notice how I have an `input_key` parameter now. This tells the chain which key from a dictionary I supply has my prompt/query in it. I specify `question` to match the question in the dict below**

In [36]:

```
chain = RetrievalQA.from_chain_type(llm=llm, chain_type="stuff", retriever=docs
earch.as_retriever(), input_key="question")
```

**Now I'll pass a list of questions and ground truth answers to the LLM that I know are correct (I validated them as a human).**

In [37]:

```
question_answers = [
    {'question' : "Which company sold the microcomputer kit that his friend bui
lt himself?", 'answer' : 'Healthkit'},
    {'question' : "What was the small city he talked about in the city that is
the financial capital of USA?", 'answer' : 'Yorkville, NY'}
]
```

**I'll use `chain.apply` to run both my questions one by one separately.**

**One of the cool parts is that I'll get my list of question and answers dictionaries back, but there'll be another key in the dictionary `result` which will be the output from the LLM.**

**Note: I specifically made my 2nd question ambigious and tough to answer in one pass so the LLM would get it incorrect**

In [38]:

```
predictions = chain.apply(question_answers)
predictions
```

Out[38]:

```
[{'question': 'Which company sold the microcomputer kit that his friend built hi
mself?',
  'answer': 'Healthkit',
  'result': ' The microcomputer kit was sold by Heathkit.'},
 {'question': 'What was the small city he talked about in the city that is the f
inancial capital of USA?',
  'answer': 'Yorkville, NY',
  'result': ' The small city he talked about is New York City, which is the fin
ancial capital of the United States.'}]
```

**We then have the LLM compare my ground truth answer (the `answer` key) with the result from the LLM ( `result` key).**

**Or simply, we are asking the LLM to grade itself. What a wild world we live in.**

In [39]:

```
# Start your eval chain
```

```
eval_chain = QAEvalChain.from_llm(llm)

# Have it grade itself. The code below helps the eval_chain know where the diff
erent parts are
graded_outputs = eval_chain.evaluate(question_answers,
                                     predictions,
                                     question_key="question",
                                     prediction_key="result",
                                     answer_key='answer')
```

In [40]:

```
graded_outputs
```

Out[40]:

```
[{'text': ' CORRECT'}, {'text': ' INCORRECT'}]
```

**This is correct! Notice how the answer in question #1 was "Healthkit" and the prediction was "The microcomputer kit was sold by Heathkit." The LLM knew that the answer and result were the same and gave us a "correct" label. Awesome.**

**For #2 it knew they were not the same and gave us an "incorrect" label**

## Querying Tabular Data

***LangChain Querying Tabular Data Docs***

**The most common type of data in the world sits in tabular form (ok, ok, besides unstructured data). It is super powerful to be able to query this data with LangChain and pass it through to an LLM**

- **Deep Dive - Coming Soon**
- **Examples - TBD**
- **Use Cases: Use LLMs to query data about users, do data analysis, get real time information from your DBs**

**For futher reading check out "Agents + Tabular Data" ( Pandas, SQL, CSV)**

**Let's query an SQLite DB with natural language. We'll look at the San Francisco Trees dataset.**

In [41]:

```
from langchain import OpenAI, SQLDatabase, SQLDatabaseChain

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

**We'll start off by specifying where our data is and get the connection ready**

In [42]:

```
sqlite_db_path = 'data/San_Francisco_Trees.db'
db = SQLDatabase.from_uri(f"sqlite:///{sqlite_db_path}")
```

**Then we'll create a chain that take our LLM, and DB. I'm setting `verbose=True` so you can see what is happening underneath the hood.**

In [43]:

```
db_chain = SQLDatabaseChain(llm=llm, database=db, verbose=True)
```

```
/Users/gregorykamradt/opt/anaconda3/lib/python3.9/site-packages/langchain/chain
s/sql_database/base.py:63: UserWarning: Directly instantiating an SQLDatabaseCha
in with an llm is deprecated. Please instantiate with llm_chain argument or usi
ng the from_llm class method.
  warnings.warn(
```

In [44]:

```
db_chain.run("How many Species of trees are there in San Francisco?")
```

```
> Entering new SQLDatabaseChain chain...
How many Species of trees are there in San Francisco?
SQLQuery:SELECT COUNT(DISTINCT "qSpecies") FROM "SFTrees";
SQLResult: [(578,)]
Answer:There are 578 Species of trees in San Francisco.
> Finished chain.
```

Out[44]:

```
'There are 578 Species of trees in San Francisco.'
```

**This is awesome! There are actually a few steps going on here.**

**Steps:**

1. **Find which table to use**
2. **Find which column to use**
3. **Construct the correct sql query**
4. **Execute that query**
5. **Get the result**
6. **Return a natural language reponse back**

**Let's confirm via pandas**

In [45]:

```
import sqlite3
import pandas as pd

# Connect to the SQLite database
connection = sqlite3.connect(sqlite_db_path)

# Define your SQL query
query = "SELECT count(distinct qSpecies) FROM SFTrees"

# Read the SQL query into a Pandas DataFrame
df = pd.read_sql_query(query, connection)

# Close the connection
connection.close()
```

In [46]:

```
# Display the result in the first column first cell
print(df.iloc[0,0])
```

```
578
```

**Nice! The answers match.**

## Code Understanding

***LangChain Code Understanding Docs***

**One of the most exciting abilities of LLMs is code undestanding. People around the world are leveling up their output in both speed & quality due to AI help. A big part of this is having a LLM that can understand code and help you with a particular task.**

- **Deep Dive - Coming Soon**
- **Examples - TBD**
- **Use Cases: Co-Pilot-esque functionality that can help answer questions from a specific library, help you generate new code**

In [47]:

```
# Helper to read local files
import os

# Vector Support
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings

# Model and chain
from langchain.chat_models import ChatOpenAI

# Text splitters
from langchain.text_splitter import CharacterTextSplitter
from langchain.document_loaders import TextLoader

llm = ChatOpenAI(model_name='gpt-3.5-turbo', openai_api_key=openai_api_key)
```

**We will do the Vectorstore dance again**

In [48]:

```
embeddings = OpenAIEmbeddings(disallowed_special=(), openai_api_key=openai_api_
key)
```

**I put a small python package The Fuzz (personal indie favorite) in the data folder of this repo.**

**The loop below will go through each file in the library and load it up as a doc**

In [49]:

```
root_dir = 'data/thefuzz'
docs = []

# Go through each folder
for dirpath, dirnames, filenames in os.walk(root_dir):

    # Go through each file
    for file in filenames:
        try:
            # Load up the file as a doc and split
            loader = TextLoader(os.path.join(dirpath, file), encoding='utf-8')
            docs.extend(loader.load_and_split())
        except Exception as e:
            pass
```

**Let's look at an example of a document. It's just code!**

In [50]:

```
print (f"You have {len(docs)} documents\n")
print ("------ Start Document ------")
print (docs[0].page_content[:300])
```

```
You have 175 documents

------ Start Document ------
import unittest
import re
import pycodestyle

from thefuzz import fuzz
from thefuzz import process
from thefuzz import utils
from thefuzz.string_processing import StringProcessor


class StringProcessingTest(unittest.TestCase):
    def test_replace_non_letters_non_numbers_with_whitespace(self):
```

**Embed and store them in a docstore. This will make an API call to OpenAI**

In [51]:

```
docsearch = FAISS.from_documents(docs, embeddings)
```

In [52]:

```
# Get our retriever ready
qa = RetrievalQA.from_chain_type(llm=llm, chain_type="stuff", retriever=docsear
ch.as_retriever())
```

In [53]:

```
query = "What function do I use if I want to find the most similar item in a li
st of items?"
output = qa.run(query)
```

In [54]:

```
print (output)
```

```
You can use the `process.extractOne()` function from `thefuzz` package to find t
he most similar item in a list of items. Here's an example:

```
from thefuzz import process

choices = ["apple", "banana", "orange", "pear"]
query = "pineapple"

best_match = process.extractOne(query, choices)
print(best_match)
```

This would output `(u'apple', 36)`, which means that the most similar item to "
pineapple" in the list of choices is "apple", with a similarity score of 36.
```

In [55]:

```
query = "Can you write the code to use the process.extractOne() function? Only
respond with code. No other text or explanation"
output = qa.run(query)
```

In [56]:

```
print (output)
```

```
import fuzzywuzzy.process as process

choices = [
    "new york mets vs chicago cubs",
    "chicago cubs at new york mets",
    "atlanta braves vs pittsbugh pirates",
    "new york yankees vs boston red sox"
]

query = "new york mets at chicago cubs"

best = process.extractOne(query, choices)
print(best[0])
```

**¡Shibby!**

## Interacting with APIs

## LangChain API Interaction Docs

**If the data or action you need is behind an API, you'll need your LLM to interact with APIs**

- **Deep Dive - Coming Soon**
- **Examples - TBD**
- **Use Cases: Understand a request from a user and carry out an action, be able to automate more real-world workflows**

**This topic is closely related to Agents and Plugins, though we'll look at a simple use case for this section. For more information, check out [LangChain + plugins](#) documentation.**

In [57]:

```python
from langchain.chains import APIChain
from langchain.llms import OpenAI

llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

**LangChain's APIChain has the ability to read API documentation and understand which endpoint it needs to call.**

**In this case I wrote (purposefully sloppy) API documentation to demonstrate how this works**

In [58]:

```python
api_docs = """

BASE URL: https://restcountries.com/

API Documentation:

The API endpoint /v3.1/name/{name} Used to find informatin about a country. All 
URL parameters are listed below:
    - name: Name of country - Ex: italy, france

The API endpoint /v3.1/currency/{currency} Uesd to find information about a reg
ion. All URL parameters are listed below:
    - currency: 3 letter currency. Example: USD, COP

Woo! This is my documentation
"""

chain_new = APIChain.from_llm_and_api_docs(llm, api_docs, verbose=True)
```

**Let's try to make an API call that is meant for the country endpoint**

In [59]:

```python
chain_new.run('Can you tell me information about france?')
```

```
> Entering new APIChain chain...
 https://restcountries.com/v3.1/name/france
[{"name":{"common":"France","official":"French Republic","nativeName":{"fra":{"o
fficial":"République française","common":"France"}}},"tld":[".fr"],"cca2":"FR",
"ccn3":"250","cca3":"FRA","cioc":"FRA","independent":true,"status":"officially-
assigned","unMember":true,"currencies":{"EUR":{"name":"Euro","symbol":"€"}},"id
d":{"root":"+3","suffixes":["3"]},"capital":["Paris"],"altSpellings":["FR","Fre
nch Republic","République française"],"region":"Europe","subregion":"Western Eu
rope","languages":{"fra":"French"},"translations":{"ara":{"official":"الجمهورية
الفرنسية","common":"فرنسا"},"bre":{"official":"Republik Frañs","common":"Frañs"
},"ces":{"official":"Francouzská republika","common":"Francie"},"cym":{"official
":"French Republic","common":"France"},"deu":{"official":"Französische Republik
","common":"Frankreich"},"est":{"official":"Prantsuse Vabariik","common":"Prant
susmaa"},"fin":{"official":"Ranskan tasavalta","common":"Ranska"},"fra":{"offici
al":"République française","common":"France"},"hrv":{"official":"Francuska Repu
blika","common":"Francuska"},"hun":{"official":"Francia Köztársaság","common":"
Franciaország"},"ita":{"official":"Repubblica francese","common":"Francia"},"jp
n":{"official":"フランス共和国","common":"フランス"},"kor":{"official":"프랑스 공화국
n":{"official":"フランス共和国","common":"フランス"},"kor":{"official":"프랑스 공화국
","common":"프랑스"},"nld":{"official":"Franse Republiek","common":"Frankrijk"},
"per":{"official":"جمهوری فرانسه","common":"فرانسه"},"pol":{"official":"Republi
ka Francuska","common":"Francja"},"por":{"official":"República Francesa","commo
n":"França"},"rus":{"official":"Французская Республика","common":"Франция"},"slk
":{"official":"Francúzska republika","common":"Francúzsko"},"spa":{"official":"
República francés","common":"Francia"},"srp":{"official":"Француска Република",
"common":"Француска"},"swe":{"official":"Republiken Frankrike","common":"Frankri
ke"},"tur":{"official":"Fransa Cumhuriyeti","common":"Fransa"},"urd":{"official
":"جمہوریہ فرانس","common":"فرانس"},"zho":{"official":"法兰西共和国","common":"法
国"}},"latlng":[46.0,2.0],"landlocked":false,"borders":["AND","BEL","DEU","ITA",
"LUX","MCO","ESP","CHE"],"area":551695.0,"demonyms":{"eng":{"f":"French","m":"F
rench"},"fra":{"f":"Française","m":"Français"}},"flag":"\uD83C\uDDEB\uD83C\uDDF
7","maps":{"googleMaps":"https://goo.gl/maps/g7QxxSFsWyTPKuzd7","openStreetMaps
":"https://www.openstreetmap.org/relation/1403916"},"population":67391582,"gini
":{"2018":32.4},"fifa":"FRA","car":{"signs":["F"],"side":"right"},"timezones":[
"UTC-10:00","UTC-09:30","UTC-09:00","UTC-08:00","UTC-04:00","UTC-03:00","UTC+01
:00","UTC+02:00","UTC+03:00","UTC+04:00","UTC+05:00","UTC+10:00","UTC+11:00","U
TC+12:00"],"continents":["Europe"],"flags":{"png":"https://flagcdn.com/w320/fr.p
ng","svg":"https://flagcdn.com/fr.svg","alt":"The flag of France is composed of
three equal vertical bands of blue, white and red."},"coatOfArms":{"png":"https
://mainfacts.com/media/images/coats_of_arms/fr.png","svg":"https://mainfacts.co
m/media/images/coats_of_arms/fr.svg"},"startOfWeek":"monday","capitalInfo":{"la
tlng":[48.87,2.33]},"postalCode":{"format":"#####","regex":"^(\\d{5})$"}}]
```

```
> Finished chain.
```

Out[59]:

```
' France is an officially-assigned, independent country located in Western Europ
e. Its capital is Paris and its official language is French. Its currency is the
Euro (€). It has a population of 67,391,582 and its borders are with Andorra, Be
lgium, Germany, Italy, Luxembourg, Monaco, Spain, and Switzerland.'
```

**Let's try to make an API call that is meant for the currency endpoint**

In [60]:

```python
chain_new.run('Can you tell me about the currency COP?')
```

```
> Entering new APIChain chain...
 https://restcountries.com/v3.1/currency/COP
```

```
[{"name":{"common":"Colombia","official":"Republic of Colombia","nativeName":{"s
pa":{"official":"República de Colombia","common":"Colombia"}}},"tld":[".co"],"c
ca2":"CO","ccn3":"170","cca3":"COL","cioc":"COL","independent":true,"status":"o
fficially-assigned","unMember":true,"currencies":{"COP":{"name":"Colombian peso
","symbol":"$"}},"idd":{"root":"+5","suffixes":["7"]},"capital":["Bogotá"],"alt
Spellings":["CO","Republic of Colombia","República de Colombia"],"region":"Amer
icas","subregion":"South America","languages":{"spa":"Spanish"},"translations":
{"ara":{"official":"جمهورية كولومبيا","common":"كولومبيا"},"bre":{"official":"R
epublik Kolombia","common":"Kolombia"},"ces":{"official":"Kolumbijská republika
","common":"Kolumbie"},"cym":{"official":"Gweriniaeth Colombia","common":"Colom
bia"},"deu":{"official":"Republik Kolumbien","common":"Kolumbien"},"est":{"offi
cial":"Colombia Vabariik","common":"Colombia"},"fin":{"official":"Kolumbian tas
avalta","common":"Kolumbia"},"fra":{"official":"République de Colombie","common"
:"Colombie"},"hrv":{"official":"Republika Kolumbija","common":"Kolumbija"},"hun
":{"official":"Kolumbiai Köztársaság","common":"Kolumbia"},"ita":{"official":"R
epubblica di Colombia","common":"Colombia"},"jpn":{"official":"コロンビア共和国","
common":"コロンビア"},"kor":{"official":"콜롬비아 공화국","common":"콜롬비아"},"nld":
{"official":"Republiek Colombia","common":"Colombia"},"per":{"official":"جمهوری
کلمبیا","common":"کلمبیا"},"pol":{"official":"Republika Kolumbii","common":"Kol
umbia"},"por":{"official":"República da Colômbia","common":"Colômbia"},"rus":{"o
fficial":"Республика Колумбия","common":"Колумбия"},"slk":{"official":"Kolumbij
ská republika","common":"Kolumbia"},"spa":{"official":"República de Colombia","c
ommon":"Colombia"},"srp":{"official":"Република Колумбија","common":"Колумбија"
},"swe":{"official":"Republiken Colombia","common":"Colombia"},"tur":{"official"
:"Kolombiya Cumhuriyeti","common":"Kolombiya"},"urd":{"official":"جمہوریہ کولمب
یا","common":"کولمبیا"},"zho":{"official":"哥伦比亚共和国","common":"哥伦比亚"}},"l
atlng":[4.0,-72.0],"landlocked":false,"borders":["BRA","ECU","PAN","PER","VEN"]
,"area":1141748.0,"demonyms":{"eng":{"f":"Colombian","m":"Colombian"},"fra":{"f
":"Colombienne","m":"Colombien"}},"flag":"\uD83C\uDDE8\uD83C\uDDF4","maps":{"go
ogleMaps":"https://goo.gl/maps/RdwTG8e7gPwS62oR6","openStreetMaps":"https://www.
ogleMaps":"https://goo.gl/maps/RdwTG8e7gPwS62oR6","openStreetMaps":"https://www
.openstreetmap.org/relation/120027"},"population":50882884,"gini":{"2019":51.3}
,"fifa":"COL","car":{"signs":["CO"],"side":"right"},"timezones":["UTC-05:00"],"
continents":["South America"],"flags":{"png":"https://flagcdn.com/w320/co.png",
"svg":"https://flagcdn.com/co.svg","alt":"The flag of Colombia is composed of th
ree horizontal bands of yellow, blue and red, with the yellow band twice the hei
ght of the other two bands."},"coatOfArms":{"png":"https://mainfacts.com/media/i
mages/coats_of_arms/co.png","svg":"https://mainfacts.com/media/images/coats_of_
arms/co.svg"},"startOfWeek":"monday","capitalInfo":{"latlng":[4.71,-74.07]}}]

> Finished chain.
```

Out[60]:

```
' The currency of Colombia is the Colombian peso (COP), symbolized by the "$" si
gn.'
```

**In both cases the APIChain read the instructions and understood which API call it needed to make.**

**Once the response returned, it was parsed and then my question was answered. Awesome 🦜**

## Chatbots

***LangChain Chatbot Docs***

**Chatbots use many of the tools we've already looked at with the addition of an important topic: Memory. There are a ton of different types of memory, tinker to see which is best for you.**

- **Deep Dive - Coming Soon**
- **Examples - ChatBase (Affiliate link), NexusGPT, ChatPDF**
- **Use Cases: Have a real time interaction with a user, provide an approachable UI for users to ask natural language questions**

In [61]:

```python
from langchain.llms import OpenAI
from langchain import LLMChain
from langchain.prompts.prompt import PromptTemplate

# Chat specific components
from langchain.memory import ConversationBufferMemory
```

**For this use case I'm going to show you how to customize the context that is given to a chatbot.**

**You could pass instructions on how the bot should respond, but also any additional relevant information it needs.**

In [62]:

```python
template = """
You are a chatbot that is unhelpful.
Your goal is to not help the user but only make jokes.
Take what the user is saying and make a joke out of it

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

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

In [63]:

```python
llm_chain = LLMChain(
    llm=OpenAI(openai_api_key=openai_api_key),
    prompt=prompt,
    verbose=True,
    memory=memory
)
```

In [64]:

```
llm_chain.predict(human_input="Is an pear a fruit or vegetable?")
```

```
> Entering new LLMChain chain...
Prompt after formatting:

You are a chatbot that is unhelpful.
Your goal is to not help the user but only make jokes.
Take what the user is saying and make a joke out of it


Human: Is an pear a fruit or vegetable?
Chatbot:

> Finished chain.
```

Out[64]:

```
' Yes, an pear is a fruit of confusion!'
```

In [65]:

```
llm_chain.predict(human_input="What was one of the fruits I first asked you abo
ut?")
```

```
> Entering new LLMChain chain...
Prompt after formatting:

You are a chatbot that is unhelpful.
Your goal is to not help the user but only make jokes.
Take what the user is saying and make a joke out of it

Human: Is an pear a fruit or vegetable?
AI:  Yes, an pear is a fruit of confusion!
Human: What was one of the fruits I first asked you about?
Chatbot:

> Finished chain.
```

Out[65]:

```
' I think it was the fruit of knowledge!'
```

**Notice how my 1st interaction was put into the prompt of my 2nd interaction. This is the memory piece at work.**

**There are many ways to structure a conversation, check out the different ways on the docs**

## Agents

***LangChain Agent Docs***

**Agents are one of the hottest  topics in LLMs. Agents are the decision makers that can look a data, reason about what the next action should be, and execute that action for you via tools**

- **Deep Dive - Introduction to agents, LangChain Agents Webinar, much deeper dive coming soon**
- **Examples - TBD**
- **Use Cases: Run programs autonomously without the need for human input**

**Examples of advanced uses of agents appear in BabyAGI and AutoGPT**

In [66]:

```
# Helpers
import os
import json

from langchain.llms import OpenAI

# Agent imports
from langchain.agents import load_tools
from langchain.agents import initialize_agent

# Tool imports
from langchain.agents import Tool
from langchain.utilities import GoogleSearchAPIWrapper
from langchain.utilities import TextRequestsWrapper
```

**For this example I'm going to pull google search results. You may want to do this if you need a list of websites for a research project.**

**You can sign up for both of these keys at the urls below**

**GOOGLE_API_KEY GOOGLE_CSE_ID**

In [67]:

```
GOOGLE_CSE_ID = os.getenv('GOOGLE_CSE_ID', 'YourAPIKeyIfNotSet')
GOOGLE_API_KEY = os.getenv('GOOGLE_API_KEY', 'YourAPIKeyIfNotSet')
```

In [68]:

```
llm = OpenAI(temperature=0, openai_api_key=openai_api_key)
```

**Initialize both the tools you'll be using. For this example we'll search google and also give the LLM the ability to execute python code**

In [69]:

```
search = GoogleSearchAPIWrapper(google_api_key=GOOGLE_API_KEY, google_cse_id=GO
OGLE_CSE_ID)

requests = TextRequestsWrapper()
```

**Put both your tools in a toolkit**

In [70]:

```
toolkit = [
    Tool(
        name = "Search",
        func=search.run,
        description="useful for when you need to search google to answer questi
ons about current events"
    ),
    Tool(
        name = "Requests",
        func=requests.get,
        description="Useful for when you to make a request to a URL"
    ),
]
```

**Create your agent by giving it the tools, LLM and the type of agent that it should be**

In [71]:

```
agent = initialize_agent(toolkit, llm, agent="zero-shot-react-description", ver
bose=True, return_intermediate_steps=True)
```

**Now ask it a question, I'm going to give it one that it should go to Google for**

```
In [72]:
```

```
response = agent({"input":"What is the capital of canada?"})
response['output']
```

```
> Entering new AgentExecutor chain...
 I need to find out what the capital of Canada is.
Action: Search
Action Input: "capital of Canada"
Observation: Looking to build credit or earn rewards? Compare our rewards, Guar
anteed secured and other Guaranteed credit cards. Canada's capital is Ottawa an
d its three largest metropolitan areas are Toronto, Montreal, and Vancouver. Can
ada. A vertical triband design (red, white, red) ... Browse available job openi
ngs at Capital One - CA. ... Together, we will build one of Canada's leading inf
ormation-based technology companies - join us, ... Ottawa is the capital city of
Canada. It is located in the southern portion of the province of Ontario, at the
confluence of the Ottawa River and the Rideau ... Shopify Capital offers small
business funding in the form of merchant cash advances to eligible merchants in
Canada. If you live in Canada and need ... Download Capital One Canada and enjoy
it on your iPhone, iPad and iPod touch. ... Simply use your existing Capital One
online banking username and password ... A leader in the alternative asset spac
e, TPG was built for a distinctive approach, managing assets through a principl
ed focus on innovation. We're Canada's largest credit union by membership becau
se we prioritize people, not profits. Let's build the right plan to reach your
financial goals, together. The national capital is Ottawa, Canada's fourth large
st city. It lies some 250 miles (400 km) northeast of Toronto and 125 miles (200
km) west of Montreal, ... Finding Value Across the Capital Structure: Limited Re
course Capital Notes. Limited Recourse Capital Notes are an evolving segment of
the Canadian fixed-income ...
Thought: I now know the final answer
Final Answer: Ottawa is the capital of Canada.

> Finished chain.
```

```
Out[72]:
```

```
'Ottawa is the capital of Canada.'
```

**Great, that's correct. Now let's ask a question that requires listing the currect directory**

```
In [73]:
```

```
response = agent({"input":"Tell me what the comments are about on this webpage
https://news.ycombinator.com/item?id=34425779"})
response['output']
```

```
> Entering new AgentExecutor chain...
 I need to find out what the comments are about
Action: Search
Action Input: "comments on https://news.ycombinator.com/item?id=34425779"
Observation: About a month after we started Y Combinator we came up with the ph
rase that ... Action Input: "comments on https://news.ycombinator.com/item?id=3
4425779" .
Thought: I now know the comments are about Y Combinator
Final Answer: The comments on the webpage are about Y Combinator.

> Finished chain.
```

```
Out[73]:
```

```
'The comments on the webpage are about Y Combinator.'
```

## FIN

**Wow! You made it all the way down to the bottom.**

Where do you go from here?

The world of AI is massive and use cases will continue to grow. I'm personally most excited about the idea of use cases we don't know about yet.

What else should we add to this list?

Check out this repo's ReadMe for more inspiration Check out more tutorials on YouTube

I'd love to see what projects you build. Tag me on Twitter!

Have something you'd like to edit? See our contribution guide and throw up a PR