LangChain Cookbook 👩‍🍳🧑‍🍳

This cookbook is based off the LangChain Conceptual Documentation (https://docs.langchain.com/docs/)

Goal: Provide an introductory understanding of the components and use cases of LangChain via ELI5 (https://www.dictionary.com/e/slang/eli5/#:~:text=ELI5%20is%20short%20for%20%E2%80%9CExplain,a%20c examples and code snippets. For use cases check out part 2 (https://github.com/gkamradt/langchain-tutorials/blob/main/LangChain%20Cookbook%20Part%202%20-%20Use%20Cases.ipynb). See video tutorial (https://www.youtube.com/watch?v=2xxziIWmaSA) of this notebook.

Links:

- LC Conceptual Documentation (https://docs.langchain.com/docs/)
- LC Python Documentation (https://python.langchain.com/en/latest/)
- LC Javascript/Typescript Documentation (https://js.langchain.com/docs/)
- LC Discord (https://discord.gg/6adMQxSpJS)
- www.langchain.com (https://langchain.com/)
- LC Twitter (https://twitter.com/LangChainAI)

What is LangChain?

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

~~**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 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 (https://docs.langchain.com/docs/)

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 (https://chat.openai.com/).

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

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 (https://platform.openai.com/docs/guides/chat/introduction)

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 n
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 t
        HumanMessage(content="I like tomatoes, what should I eat?")
    ]
)
```

Out[4]:
```
AIMessage(content='You could try a caprese salad with fresh tomatoes, mozzarella, and b
asil.')
```

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
        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 ind
ulge 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 o
         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
other places", metadata={'my_document_id': 234234, 'my_document_source': 'The LangChain
Papers', '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 o
```

Out[9]:
```
Document(page_content="This is my document. It is full of text that I've gathered from
other 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 (https://platform.openai.com/docs/models)

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
        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 proble
ms and needed a change of scenery!')
```

Function Calling Models

Function calling models (https://openai.com/blog/function-calling-and-other-api-updates) 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_k

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_weathe
r', '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, -0.0031846734422911363, -0.00077342069146477
14, -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 (https://realpython.com/python-f-strings/) in python but for prompts

Advanced: Check out LangSmithHub(https://smith.langchain.com/hub (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 (https://python.langchain.com/docs/modules/model_io/prompts/example_selectors/)

If you want an overview on why examples are important (prompt engineering), check out this video (https://www.youtube.com/watch?v=dOxUroR57xs)

In [20]:
```python
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_lat
est_version.py:32: UserWarning: A newer version of deeplake (3.7.2) is available. It's
recommended that you update to the latest version using `pip install -U deeplake`.
  warnings.warn(
```

In [21]:
```python
# SemanticSimilarityExampleSelector will select examples that are similar to your input

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 s
    OpenAIEmbeddings(openai_api_key=openai_api_key),

    # This is the VectorStore class that is used to store the embeddings and do a simila
    Chroma,

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

In [22]:
```python
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 (https://python.langchain.com/docs/modules/model_io/output_parsers).

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
    ResponseSchema(name="good_string", description="This is your response, a reformatted
]

# 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, includi
ng 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, includi
ng 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 Cal
ifornia!"\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 (https://platform.openai.com/docs/guides/gpt/function-calling) for other methods.

In order to use this method you'll need to use a model that supports function calling (https://openai.com/blog/function-calling-and-other-api-updates#:~:text=Developers%20can%20now%20describe%20functions%20to%20gpt%2D4%2D0613%20and) I'll use `gpt4-0613`

Example 1: Simple

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
)
```

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
)
```

Out[35]:
```
People(people=[Person(name='Sally', age=13, fav_food=None), Person(name='Joey', age=12,
fav_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 te
```

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
)
```

Out[38]:
```
Products(products=[<Product.CRM: 'CRM'>, <Product.HARDWARE: 'HARDWARE'>, <Product.VIDEO
_EDITING: '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 (https://openai.com/blog/chatgpt-plugins) specifically retrieval plugins (https://github.com/openai/chatgpt-retrieval-plugin)

See a big list (https://python.langchain.com/en/latest/modules/indexes/document_loaders.html) of document loaders here. A bunch more on Llama Index (https://llamahub.ai/) 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)
are very Ozzie_osman 8 months ago
             | parent | next [-]

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

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 (http://www.paulgraham.com/)

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 star
tup? 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 (https://python.langchain.com/en/latest/modules/indexes/text_splitters.html) 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]:
```
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 (https://python.langchain.com/docs/modules/data_connection/document_transformers/)

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

In [54]:
```
retriever
```

Out[54]:
```
VectorStoreRetriever(tags=['FAISS'], vectorstore=<langchain.vectorstores.faiss.FAISS ob
ject at 0x7f8389169070>)
```

In [55]:
```
docs = retriever.get_relevant_documents("what types of things did the author want to bui
```

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 thing
s, 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, th
eory
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 (https://www.pinecone.io/) & Weaviate (https://weaviate.io/). More examples on OpenAIs retriever documentation (https://github.com/openai/chatgpt-retrieval-plugin#choosing-a-vector-database). Chroma (https://www.trychroma.com/) & FAISS (https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/) 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, -0.01287480474626
6883]...
```

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 (https://python.langchain.com/en/latest/modules/memory/how_to_guides.html)](https://python.langchain.com/en/latest/modules/memory/how_to_guides.html) 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 (https://www.youtube.com/watch?v=f9_BWhCI4Zo&t=2s) explaining different summarization chain types

There are many applications of chains (https://python.langchain.com/en/latest/modules/chains/how_to_guides.html) 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 su
% 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 h
% 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, b
lack 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 packa
ge directions.
2. Meanwhile, add the olive oil to a large skillet over medium-high heat. Add the diced
pancetta 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. R
emove 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 (https://www.youtube.com/watch?v=f9_BWhCI4Zo)](https://www.youtube.com/watch?v=f9_BWhCI4Zo) for other chain types besides map-reduce

In [70]:

```python
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
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 wr
ong impression that they never faced any risks to achieve successful results. An exampl
e of this is Newton, whose smartness is assumed to have led straight him to truths with
out any detours into alchemy or theology - despite the fact that he spent a lot of time
on both fields. Maybe the simpler explanation is that he was willing to take risks, eve
n if it means potentially making mistakes.

 In the 17th century, Newton took a risk and made three bets, one of which turned out t
o be a successful invention of what we now call physics. The other two bets were on les
s popular subjects of the time such as alchemy and theology. People did not know then w
hat the 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 w
as willing to take risks and explore multiple fields to make his discoveries. He placed
three risky bets, one of which resulted in the creation of physics as we know it toda
y."

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 (https://www.youtube.com/watch?v=L_Guz73e6fw&t=867s)'. 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 (https://python.langchain.com/en/latest/modules/agents/agents/agent_types.html).

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 (https://platform.openai.com/docs/plugins/introduction).

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")
```

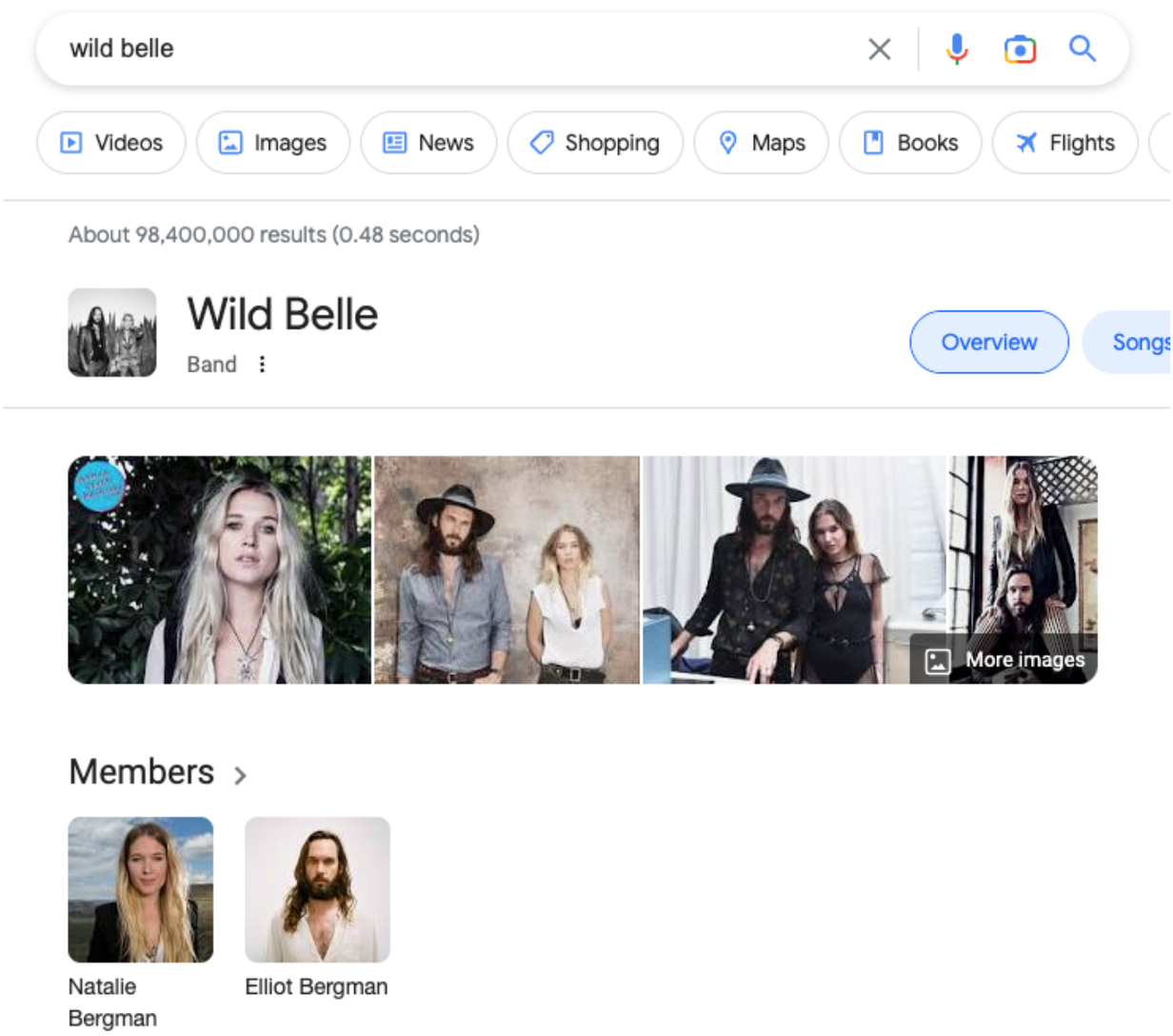
In [73]:
```
toolkit = load_tools(["serpapi"], llm=llm, serpapi_api_key=serpapi_api_key)
```

In [74]:
```
agent = initialize_agent(toolkit, llm, agent="zero-shot-react-description", verbose=True
```

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
duo Wild Belle, along with her brother Elliot Bergman. Her debut solo album, Mercy, was
released on Third Man Records on May 7, 2021. She is based in Los Angeles.', 'Natalie B
ergman type: American singer-songwriter.', 'Natalie Bergman main_tab_text: Overview.',
'Natalie Bergman kgmid: /m/0qgx4kh.', 'Natalie Bergman genre: Folk.', 'Natalie Bergman
parents: Susan Bergman, Judson Bergman.', 'Natalie Bergman born: 1988 or 1989 (age 34-3
5).', 'Natalie Bergman is an American singer-songwriter. She is one half of the duo Wil
d Belle, along 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 (https://open.spotify.com/track/1eREJIBdqeCcqNCB1pbz7w?si=c014293b63c7478c)