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 (coming soon).

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

```
openai_api_key='YourAPIKey'
```

LangChain Components

Schema - Nuts and Bolts of working with LLMs

Text

The natural language way to interact with LLMs

```
# You'll be working with simple strings (that'll soon grow in complexity!)
my_text = "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

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

chat = ChatOpenAI(temperature=.7, openai_api_key=openai_api_key)

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?")
    ]
)
```

```
AIMessage(content='You could try making a tomato salad with fresh basil and mozzarella cheese.', additional_kwargs={})
```

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

```
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?")
    ]
)
```

```
AIMessage(content='While in Nice, you can also explore the charming
old town, visit the famous Promenade des Anglais, and indulge in
delicious French cuisine.', additional_kwargs={})
```

Documents

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

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

```
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
         })
```

```
Document(page_content="This is my document. It is full of text that
I've gathered from other places", lookup_str='',
metadata={'my_document_id': 234234, 'my_document_source': 'The
LangChain Papers', 'my_document_create_time': 1680013019},
lookup_index=0)
```

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. See more models here

```
from langchain.llms import OpenAI

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

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

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

Chat Model

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

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

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

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?")
    ]
)
```

```
AIMessage(content="You could try walking, but I don't recommend it
unless you have a lot of time on your hands. Maybe try flapping your
arms really hard and see if you can fly there?", additional_kwargs={})
```

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.'

```
from langchain.embeddings import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)

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

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

```
Your embedding is length 1536
Here's a sample: [-0.00020583387231454253, -0.003205398330464959, -
0.0008301587076857686, -0.01946892775595188, -0.015162716619670391]...
```

Prompts - Text generally used as instructions to your model

Prompt

What you'll pass to the underlying model

```
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?
"""
```

```
llm(prompt)
```

```
'\nThe 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

```
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 Pantheon, and the Trevi Fountain for a taste
of Rome's ancient and modern culture.
```

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

```
from langchain.prompts.example_selector import
SemanticSimilarityExampleSelector
from langchain.vectorstores import FAISS
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"},
]

# SemanticSimilarityExampleSelector will select examples that are
similar to your input by 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 semantic similarity.
    OpenAIEmbeddings(openai_api_key=openai_api_key),

    # This is the VectorStore class that is used to store the
embeddings and do a similarity search over.
    FAISS,
```

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

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"],
)

# Select a noun!
my_noun = "student"

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

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

Example Input: driver
Example Output: car

Example Input: pilot
Example Output: plane

Input: student
Output:
```

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

```
' classroom'
```

Output Parsers

A helpful way to format the output of a model. Usually used for structured output.

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)

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

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

# How you would like your reponse structured. This is basically a
fancy prompt template
response_schemas = [
    ResponseSchema(name="bad_string", description="This a poorly
formatted user input string"),
    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)

# 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:

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

```python
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:

```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:
```

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

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

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

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

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.

```
from langchain.document_loaders import HNLoader

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

data = loader.load()

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:

dang 69 days ago
             | next [–]

Related ongoing thread:GPT-3.5 and Wolfram Alpha via LangChain - https://news.ycombinator.com/item?id=3440ozzie_osman 69 days ago
             | prev | next [–]

LangChain is awesome. For people not sure what it's doing, large language models (LLMs) are
```

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.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# 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
```

```
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])

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

```
You have 606 documents
```

```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,
```

Retrievers

Easy way to combine documents with language models.

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

```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()

# 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)

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

```python
retriever
```

```
VectorStoreRetriever(vectorstore=<langchain.vectorstores.faiss.FAISS
object at 0x7fb81007a9d0>, search_type='similarity', search_kwargs={})
```

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

```python
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, theory
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}

```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()

# 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)

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

```
You have 78 documents
```

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

```
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: [-0.001125750364735722 5, -0.01111479103565216,
-0.012860921211540699]...
```

Your vectorstore store your embeddings (☝️) and make the 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

```
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?")

history.messages
```

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

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

```
AIMessage(content='The capital of France is Paris.',
additional_kwargs={})
```

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

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

```
 AIMessage(content='The capital of France is Paris.', additional_kwargs={})]
```

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 LMM as an input into another. Good for breaking up tasks (and keeping your LLM focused)

```python
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)

template = """Your job is to come up with a classic dish from the area that the users suggests.
% 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)

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

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

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

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

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

```
> Entering new SimpleSequentialChain chain...
A classic dish from Rome is Spaghetti alla Carbonara, a pasta dish
made with egg, cheese, guanciale (cured pork cheek), and black pepper.

Ingredients:
-1/2 lb. spaghetti
-4 oz. guanciale, diced
-2 cloves garlic, minced
-2 eggs
-2/3 cup Parmigiano Reggiano cheese, divided
-1/4 tsp. freshly cracked black pepper
-1/4 cup reserved pasta water
-2 tablespoons olive oil
-Parsley for garnish (optional)

Instructions:
1. Boil spaghetti in a large pot of salted boiling water until al
dente, about 8 minutes. Reserve 1/4 cup of cooking water and drain the
spaghetti.

2. In a large skillet, heat oil over medium-high heat, then add
guanciale and sauté until lightly brown, about 5 minutes.

3. Add garlic and sauté for an additional 1-2 minutes.

4. In a medium bowl, whisk together eggs and 1/3 cup Parmigiano
Reggiano cheese.

5. Add cooked spaghetti to the large skillet, toss to combine, then
reduce the heat to medium-low.

6. Pour in the egg and cheese mixture, then add pepper and reserved
pasta water.

7. Toss pasta

> 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

```
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...
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:


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


" Biographies of famous scientists often omit the risks they took and
the mistakes they made during their lifetime. This gives us an
impression that these scientists had a perfect judgement, when in fact
they made unwise decisions like Newton's dabblings in alchemy and
theology. Perhaps these scientists were just taking risks and making
mistakes like anyone else.


This passage discusses how, in the time of Sir Isaac Newton, the three
areas of study – physics, alchemy, and theology – were all considered
equally valuable and worthy of exploration. Newton's success in the
area of physics has since made the others seem like a waste of time,
however at the point of Newton's exploration, all three were seen as
high-risk but high-reward propositions."


CONCISE SUMMARY:

> Finished chain.

> Finished chain.
```

```
" Biographies of famous scientists often omit the risks they took and mistakes they made, creating an impression of perfect judgement. Sir Isaac Newton's exploration of physics, alchemy, and theology was seen as all high-risk but high-reward propositions at the time, and should not be overlooked."
```

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:

```python
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)

serpapi_api_key='...'
```

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

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

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.
Thought: I should search for the debut album of Wild Belle.
Action: Search
Action Input: "Wild Belle debut album"
Observation: Isles
Thought: I now know the final answer.
Final Answer: Isles is the debut album of Wild Belle, the band that
Natalie Bergman is a part of.

> Finished chain.
```

```
print(json.dumps(response["intermediate_steps"], indent=2))
```


```
[
  [
    [
      "Search",
      "Natalie Bergman band",
      " I should try to find out what band Natalie Bergman is a part
of.\nAction: Search\nAction Input: \"Natalie Bergman band\""
    ],
    "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."
  ],
  [
    [
      "Search",
      "Wild Belle debut album",
      " I should search for the debut album of Wild Belle.\nAction:
Search\nAction Input: \"Wild Belle debut album\""
    ],
```

```
    "Isles"
  ]
]
```

Wild Belle

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