

LangChain for LLM Application Development

LangChain: Models, Prompts and Output Parsers

- Direct API calls to OpenAI
- API calls through LangChain:
 - Prompts
 - Models
 - Output parsers

其实agent就是这样，制定按thought、actions、observation格式输出，并且是step by step 的思维链

EXAMPLES = [""

Question: What is the elevation range for the area that the eastern sector of the Colorado orogeny extends into?

Thought: need to search Colorado orogeny, find the area that the eastern sector of the Colorado orogeny extends into, then find the elevation range of the area.

Action: Search[Colorado orogeny]

Observation: The Colorado orogeny was an episode of mountain building (an orogeny) in Colorado and surrounding areas.

Thought: It does not mention the eastern sector. So I need to look up eastern sector.
Action: Lookup[eastern sector]

...

Thought: High Plains rise in elevation from around 1,800 to 7,000 ft, so the answer is 1,800 to 7,000 ft.

Action: Finish[1,800 to 7,000 ft]""

LangChain library functions parse the LLM's output assuming that it will use certain keywords.

Example here uses **Thought, Action, Observation** as keywords for Chain-of-Thought Reasoning. (ReAct)

Prompts Template

```
from langchain.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate

chat = ChatOpenAI(temperature=0.0)
```

```

template_string = """Translate the text \
that is delimited by triple backticks \
into a style that is {style}. \
text: ```{text}```\
"""
prompt_template = ChatPromptTemplate.from_template(template_string)

```

Output Parsers

指定输出格式 ⇒ instruction prompt ⇒ get response ⇒ parse

```

from langchain.output_parsers import ResponseSchema
from langchain.output_parsers import StructuredOutputParser

gift_schema = ResponseSchema(name="gift",
                             description="Was the item purchased\
as a gift for someone else? \
Answer True if yes,\
False if not or unknown.")
delivery_days_schema = ResponseSchema(name="delivery_days",
                                       description="How many days\
did it take for the product\
to arrive? If this \
information is not found,\
output -1.")
price_value_schema = ResponseSchema(name="price_value",
                                     description="Extract any\
sentences about the value or \
price, and output them as a \
comma separated Python list.")

response_schemas = [gift_schema,
                     delivery_days_schema,
                     price_value_schema]

# 1.
output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
# 2.
format_instructions = output_parser.get_format_instructions()
# print
The output should be a markdown code snippet formatted in the following schema, including the leading and trailing ```json and ```

```json
{
 "gift": string // Was the item purchased as a gift for someone else? Answer Tru
 "delivery_days": string // How many days did it take for the product
 "price_value": string // Extract any sentences about the value or
}
review_template_2 = """\
For the following text, extract the following information:

gift: Was the item purchased as a gift for someone else? \
Answer True if yes, False if not or unknown.

delivery_days: How many days did it take for the product\
to arrive? If this information is not found, output -1.

price_value: Extract any sentences about the value or price,\
and output them as a comma separated Python list.

text: {text}

{format_instructions}
"""
response = chat(messages)
output_dict = output_parser.parse(response.content)

```

## LangChain: Memory

- ConversationBufferMemory

- ConversationBufferWindowMemory
- ConversationTokenBufferMemory
- ConversationSummaryMemory

## ConversationBufferMemory

没有限制的内存

## ConversationBufferWindowMemory

限制对话次数

```
memory = ConversationBufferWindowMemory(k=1)
```

## ConversationTokenBufferMemory

限制总的token长度，会保证对话完整且总的token不超过limit

```
memory = ConversationTokenBufferMemory(llm=llm, max_token_limit=30)
```

## ConversationSummaryMemory

最有用，限制token limit，会通过summary来保证不超过总的长度；

summary也是用llm生成的，如果数据量非常大，还是要上vector store

```
create a long string
schedule = "There is a meeting at 8am with your product team. \
You will need your powerpoint presentation prepared. \
9am-12pm have time to work on your LangChain \
project which will go quickly because Langchain is such a powerful tool. \
At Noon, lunch at the italian restaurant with a customer who is driving \
from over an hour away to meet you to understand the latest in AI. \
Be sure to bring your laptop to show the latest LLM demo."

memory = ConversationSummaryBufferMemory(llm=llm, max_token_limit=100)
memory.save_context({"input": "Hello"}, {"output": "What's up"})
memory.save_context({"input": "Not much, just hanging"}, {"output": "Cool"})
memory.save_context({"input": "What is on the schedule today?"}, {"output": f"{schedule}"})

conversation = ConversationChain(
 llm=llm,
 memory = memory,
 verbose=True
)
conversation.predict(input="What would be a good demo to show?")

output
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context.

Current conversation:
System: The human and AI engage in small talk before discussing the day's schedule. The AI informs the human of a morning meeting with the Human: What would be a good demo to show?
AI:
```

# Memory Types

## ConversationBufferMemory

- This memory allows for storing of messages and then extracts the messages in a variable.

## ConversationBufferWindowMemory

- This memory keeps a list of the interactions of the conversation over time. It only uses the last K interactions.

## ConversationTokenBufferMemory

- This memory keeps a buffer of recent interactions in memory, and uses token length rather than number of interactions to determine when to flush interactions.

## ConversationSummaryMemory

- This memory creates a summary of the conversation over time.

# Additional Memory Types

## Vector data memory

- Stores text (from conversation or elsewhere) in a vector database and retrieves the most relevant blocks of text.

## Entity memories

- Using an LLM, it remembers details about specific entities.

You can also use multiple memories at one time.

E.g., Conversation memory + Entity memory to recall individuals.

You can also store the conversation in a conventional database (such as key-value store or SQL)

内存历史 ⇒ summary ⇒ vector store ⇒ k-v

---

## Chains in LangChain

- LLMChain
- Sequential Chains
  - SimpleSequentialChain
  - SequentialChain
- Router Chain

## LLMChain

最基本的，llm + prompt

```
chain = LLMChain(llm=llm, prompt=prompt)
```

## Sequential Chains

将多个llmchain组合，simple是单向输入单向输出，sequential是多个输入多个输出

```
prompt template 2
second_prompt = ChatPromptTemplate.from_template(
 "Write a 50 words description for the following \
 company:{company_name}"
)
chain 2
chain_two = LLMChain(llm=llm, prompt=second_prompt)
simple
overall_simple_chain = SimpleSequentialChain(chains=[chain_one, chain_two],
 verbose=True
)

sequential
必须要精确指定每一个chain输入输出的key，这样才知道走哪个chain

llm = ChatOpenAI(temperature=0.9)

prompt template 1: translate to english
first_prompt = ChatPromptTemplate.from_template(
 "Translate the following review to english:"
 "\n\n{Review}"
)
chain 1: input= Review and output= English_Review
chain_one = LLMChain(llm=llm, prompt=first_prompt,
 output_key="English_Review"
)

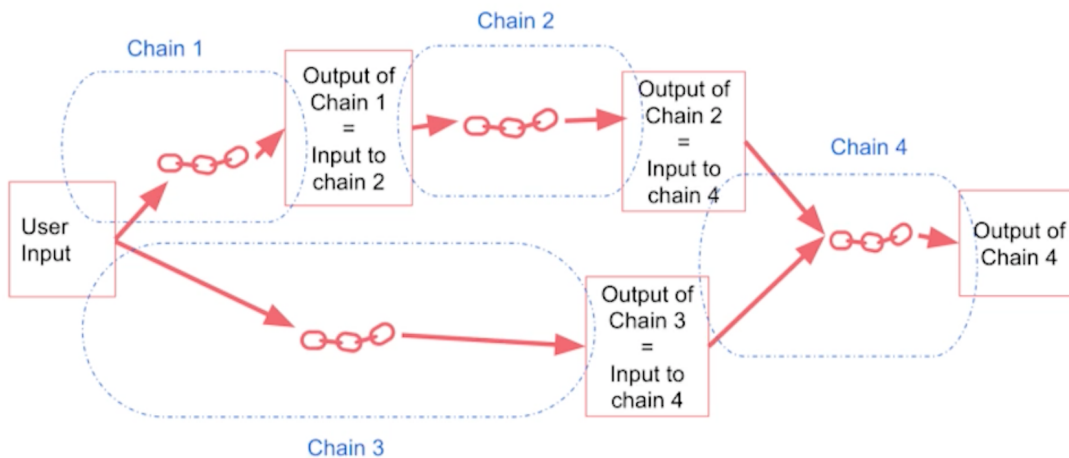
second_prompt = ChatPromptTemplate.from_template(
 "Can you summarize the following review in 1 sentence:"
 "\n\n{English_Review}"
)
chain 2: input= English_Review and output= summary
chain_two = LLMChain(llm=llm, prompt=second_prompt,
 output_key="summary"
)

prompt template 3: translate to english
third_prompt = ChatPromptTemplate.from_template(
 "What language is the following review:\n\n{Review}"
)
chain 3: input= Review and output= language
chain_three = LLMChain(llm=llm, prompt=third_prompt,
 output_key="language"
)

prompt template 4: follow up message
fourth_prompt = ChatPromptTemplate.from_template(
 "Write a follow up response to the following "
 "summary in the specified language:"
 "\n\nSummary: {summary}\n\nLanguage: {language}"
)
chain 4: input= summary, language and output= followup_message
chain_four = LLMChain(llm=llm, prompt=fourth_prompt,
 output_key="followup_message"
)

overall_chain: input= Review
and output= English_Review,summary, followup_message
overall_chain = SequentialChain(
 chains=[chain_one, chain_two, chain_three, chain_four],
 input_variables=["Review"],
 output_variables=["English_Review", "summary", "followup_message"],
 verbose=True
)

注意template中的输入变量，和llm中的ouputkey
```



## Router Chain

当chain数量类型很多时候，就不能够靠制定输入输出key了，而是通过一个router来让llm决定走哪个chain，简单的agent思想

```

from langchain.chains.router import MultiPromptChain
from langchain.chains.router.llm_router import LLMRouterChain, RouterOutputParser
from langchain.prompts import PromptTemplate

llm = ChatOpenAI(temperature=0)

destination_chains = {}
for p_info in prompt_infos:
 name = p_info["name"]
 prompt_template = p_info["prompt_template"]
 prompt = ChatPromptTemplate.from_template(template=prompt_template)
 chain = LLMChain(llm=llm, prompt=prompt)
 destination_chains[name] = chain

destinations = [f"{p['name']}: {p['description']}" for p in prompt_infos]
destinations_str = "\n".join(destinations)

default_prompt = ChatPromptTemplate.from_template("{input}")
default_chain = LLMChain(llm=llm, prompt=default_prompt)

MULTI_PROMPT_ROUTER_TEMPLATE = """Given a raw text input to a \
language model select the model prompt best suited for the input. \
You will be given the names of the available prompts and a \
description of what the prompt is best suited for. \
You may also revise the original input if you think that revising\
it will ultimately lead to a better response from the language model.

<< FORMATTING >>
Return a markdown code snippet with a JSON object formatted to look like:
```json
{{{
  "destination": string \ name of the prompt to use or "DEFAULT"
  "next_inputs": string \ a potentially modified version of the original input
}}}}
```

```

```

REMEMBER: "destination" MUST be one of the candidate prompt \
names specified below OR it can be "DEFAULT" if the input is not\
well suited for any of the candidate prompts.
REMEMBER: "next_inputs" can just be the original input \
if you don't think any modifications are needed.

<< CANDIDATE PROMPTS >>
{destinations}

<< INPUT >>
{{input}}

<< OUTPUT (remember to include the ``json``>>""

router_template = MULTI_PROMPT_ROUTER_TEMPLATE.format(
 destinations=destinations_str
)
router_prompt = PromptTemplate(
 template=router_template,
 input_variables=["input"],
 output_parser=RouterOutputParser(),
)

router_chain = LLMRouterChain.from_llm(llm, router_prompt)

chain = MultiPromptChain(router_chain=router_chain,
 destination_chains=destination_chains,
 default_chain=default_chain, verbose=True
)

```

## LangChain: Q&A over Documents

通过index直接索引，或者自己组装chain，indexes也只是封装好的chain

直接用index chain

```

from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import CSVLoader
from langchain.vectorstores import DocArrayInMemorySearch
from IPython.display import display, Markdown
from langchain.indexes import VectorstoreIndexCreator

file = 'OutdoorClothingCatalog_1000.csv'
loader = CSVLoader(file_path=file)

index = VectorstoreIndexCreator(
 vectorstore_cls=DocArrayInMemorySearch
).from_loaders([loader])

query = "Please list all your shirts with sun protection \
in a table in markdown and summarize each one."

也就是llm chain
response = index.query(query)

```

自己组装

```

embeddings
from langchain.embeddings import OpenAIEmbeddings
embeddings = OpenAIEmbeddings()

docs
loader = CSVLoader(file_path=file)
docs = loader.load()
vector db

```



```

db = DocArrayInMemorySearch.from_documents(
 docs,
 embeddings
)
docs = db.similarity_search(query)
retriever
retriever = db.as_retriever()

llm = ChatOpenAI(temperature = 0.0)
small docs to one text
qdocs = "".join([docs[i].page_content for i in range(len(docs))])

response = llm.call_as_llm(f"{qdocs} Question: Please list all your \
shirts with sun protection in a table in markdown and summarize each one.")

封装成chain
qa_stuff = RetrievalQA.from_chain_type(
 llm=llm,
 chain_type="stuff",
 retriever=retriever,
 verbose=True
)

response = qa_stuff.run(query)

```

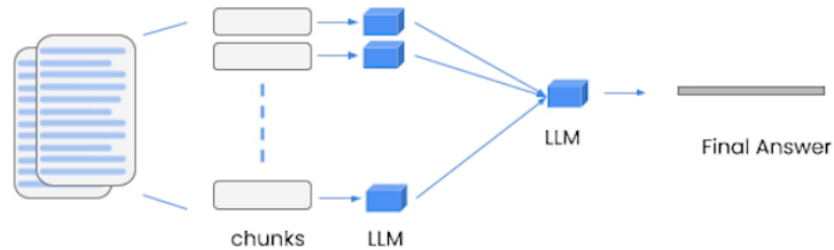
## methods to handle docs

stuff, map reduce, refine, map rerank

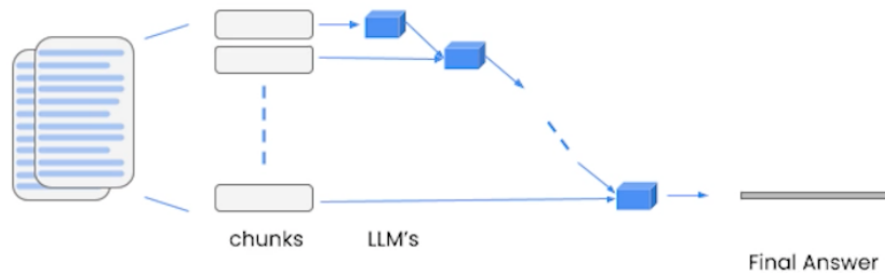
stuff最常用，就是把所有文本都拿来做事imilar\_search，其次是map reduce，分批处理大文本，摘要 ⇒ 摘要的摘要

## 3 additional methods

### 1. Map\_reduce



### 2. Refine



### 3. Map\_rerank



## LangChain: Evaluation

- Example generation
- Manual evaluation (and debugging)
- LLM-assisted evaluation

```
index = VectorstoreIndexCreator(
 vectorstore_cls=DocArrayInMemorySearch
) .from_loaders([loader])

llm = ChatOpenAI(temperature = 0.0)
qa = RetrievalQA.from_chain_type(
 llm=llm,
 chain_type="stuff",
 retriever=index.vectorstore.as_retriever(),
 verbose=True,
 chain_type_kwargs = {
```

```

 "document_separator": "<<<<>>>>"
 }
)

```

## Example generation

手动, langchain生成 examples

```

from langchain.evaluation.qa import QAGenerateChain
example_gen_chain = QAGenerateChain.from_llm(ChatOpenAI())
new_examples = example_gen_chain.apply_and_parse(
 [{"doc": t} for t in data[:5]]
)

```

## Manual evaluation (and debugging)

去观察每一步到底发生了什么来debug, chain就是函数调用, 先捕获再冒泡

```

import langchain
langchain.debug = True
qa.run(examples[0]["query"])

[chain/start] [1:chain:RetrievalQA] Entering Chain run with input:
{
 "query": "Do the Cozy Comfort Pullover Set have side pockets?"
}
[chain/start] [1:chain:RetrievalQA > 2:chain:StuffDocumentsChain] Entering Chain run with input:
[inputs]
[chain/start] [1:chain:RetrievalQA > 2:chain:StuffDocumentsChain > 3:chain:LLMChain] Entering Chain run with input:
{
 "question": "Do the Cozy Comfort Pullover Set have side pockets?",
 "context": ": 10\\nname: Cozy Comfort Pullover Set, Stripe\\ndescription: Perfect for lounging, this striped knit set lives up to its name.
}
[llm/start] [1:chain:RetrievalQA > 2:chain:StuffDocumentsChain > 3:chain:LLMChain > 4:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "System: Use the following pieces of context to answer the users question. \\nIf you don't know the answer, just say that you don't know
]
}
[llm/end] [1:chain:RetrievalQA > 2:chain:StuffDocumentsChain > 3:chain:LLMChain > 4:llm:ChatOpenAI] [1.73s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "The Cozy Comfort Pullover Set, Stripe does have side pockets.",
 "generation_info": null,
 "message": {
 "content": "The Cozy Comfort Pullover Set, Stripe does have side pockets.",
 "additional_kwargs": {},
 "example": false
 }
]
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 628,
 "completion_tokens": 14,
 "total_tokens": 642
 },
 "model_name": "gpt-3.5-turbo"
 }
}
[chain/end] [1:chain:RetrievalQA > 2:chain:StuffDocumentsChain > 3:chain:LLMChain] [1.73s] Exiting Chain run with output:
{
 "text": "The Cozy Comfort Pullover Set, Stripe does have side pockets."
}
[chain/end] [1:chain:RetrievalQA > 2:chain:StuffDocumentsChain] [1.73s] Exiting Chain run with output:
{
 "output_text": "The Cozy Comfort Pullover Set, Stripe does have side pockets."
}
[chain/end] [1:chain:RetrievalQA] [1.97s] Exiting Chain run with output:
{

```

```

 "result": "The Cozy Comfort Pullover Set, Stripe does have side pockets."
}
'The Cozy Comfort Pullover Set, Stripe does have side pockets.'
```

## LLM-assisted evaluation

```

from langchain.evaluation.qa import QAEvalChain
根据example预测结果
predictions = qa.apply(examples)

llm = ChatOpenAI(temperature=0)
eval_chain = QAEvalChain.from_llm(llm)
比较examples和predictions并打分
graded_outputs = eval_chain.evaluate(examples, predictions)
```

## LangChain: Agents

基本元素就是action, thought, observation

```

from langchain.agents.agent_toolkits import create_python_agent
from langchain.agents import load_tools, initialize_agent
from langchain.agents import AgentType
from langchain.tools.python.tool import PythonREPLTool
from langchain.python import PythonREPL
from langchain.chat_models import ChatOpenAI
from langchain.agents import tool
from datetime import date

llm = ChatOpenAI(temperature=0)
tools 可以自定义的
tools = load_tools(["llm-math", "wikipedia"], llm=llm)

@tool
def time(text: str) -> str:
 """Returns todays date, use this for any \
 questions related to knowing todays date. \
 The input should always be an empty string, \
 and this function will always return todays \
 date - any date mathematics should occur \
 outside this function."""
 return str(date.today())
tools += [time]

agent= initialize_agent(
 tools,
 llm,
 agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION, # Chat经过优化适合对话, react适合推理
 handle_parsing_errors=True,
 verbose = True)

import langchain
langchain.debug=True
agent.run(f"""find out the longest first name and shortest last name then combine it to a name
and print the output: {customer_list}""")
langchain.debug=False
```

### agent运行流程

```

[chain/start] [1:chain:AgentExecutor] Entering Chain run with input:
{
 "input": "Sort these customers by last name and then first name and print the output: [['Harrison', 'Chase'], ['Lang', 'Chain'], ['Dolly']
}
[chain/start] [1:chain:AgentExecutor > 2:chain:LLMChain] Entering Chain run with input:
{
 "input": "Sort these customers by last name and then first name and print the output: [['Harrison', 'Chase'], ['Lang', 'Chain'], ['Dolly']
```

```

"agent_scratchpad": "",
"stop": [
 "\nObservation:",
 "\n\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 2:chain:LLMChain > 3:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca
]
}
[llm/end] [1:chain:AgentExecutor > 2:chain:LLMChain > 3:llm:ChatOpenAI] [4.73s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "I can use the sorted() function to sort the list of customers by last name and then first name. I will need to provide a k
 "generation_info": null,
 "message": {
 "content": "I can use the sorted() function to sort the list of customers by last name and then first name. I will need to provid
 "additional_kwargs": {},
 "example": false
 }
 }
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 327,
 "completion_tokens": 139,
 "total_tokens": 466
 },
 "model_name": "gpt-3.5-turbo"
 }
}
[chain/end] [1:chain:AgentExecutor > 2:chain:LLMChain] [4.73s] Exiting Chain run with output:
{
 "text": "I can use the sorted() function to sort the list of customers by last name and then first name. I will need to provide a key fun
}
[tool/start] [1:chain:AgentExecutor > 4:tool:Python REPL] Entering Tool run with input:
"""
customers = [['Harrison', 'Chase'], ['Lang', 'Chain'], ['Dolly', 'Too'], ['Elle', 'Elem'], ['Geoff', 'Fusion'], ['Trance', 'Former'], ['Jen
sorted_customers = sorted(customers, key=lambda x: (x[1], x[0]))
print(sorted_customers)
"""
[tool/end] [1:chain:AgentExecutor > 4:tool:Python REPL] [0.451ms] Exiting Tool run with output:
"[['Jen', 'Ayai'], ['Lang', 'Chain'], ['Harrison', 'Chase'], ['Elle', 'Elem'], ['Trance', 'Former'], ['Geoff', 'Fusion'], ['Dolly', 'Too']]
[chain/start] [1:chain:AgentExecutor > 5:chain:LLMChain] Entering Chain run with input:
{
 "input": "Sort these customers by last name and then first name and print the output: [['Harrison', 'Chase'], ['Lang', 'Chain'], ['Dolly'
 "agent_scratchpad": "I can use the sorted() function to sort the list of customers by last name and then first name. I will need to provi
 "stop": [
 "\nObservation:",
 "\n\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 5:chain:LLMChain > 6:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca
]
}
[llm/end] [1:chain:AgentExecutor > 5:chain:LLMChain > 6:llm:ChatOpenAI] [3.86s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "The customers are now sorted by last name and then first name. The output is correct.\nFinal Answer: [['Jen', 'Ayai'], ['L
 "generation_info": null,
 "message": {
 "content": "The customers are now sorted by last name and then first name. The output is correct.\nFinal Answer: [['Jen', 'Ayai']
 "additional_kwargs": {},
 "example": false
 }
 }
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 522,

```

```

 "completion_tokens": 72,
 "total_tokens": 594
 },
 "model_name": "gpt-3.5-turbo"
}
}
[chain/end] [1:chain:AgentExecutor > 5:chain:LLMChain] [3.87s] Exiting Chain run with output:
{
 "text": "The customers are now sorted by last name and then first name. The output is correct.\nFinal Answer: [['Jen', 'Ayai'], ['Lang',
}
[chain/end] [1:chain:AgentExecutor] [8.60s] Exiting Chain run with output:
{
 "output": "[['Jen', 'Ayai'], ['Lang', 'Chain'], ['Harrison', 'Chase'], ['Elle', 'Elem'], ['Trance', 'Former'], ['Geoff', 'Fusion'], ['Dol
}

```

我换了个问题，结果他给自己绕进去了哈哈

```

import langchain
langchain.debug=True
agent.run(f"""find out the longest first name and shortest last name then combine it to a name
and print the output: {customer_list}""")
langchain.debug=False

[chain/start] [1:chain:AgentExecutor] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],
}
[chain/start] [1:chain:AgentExecutor > 2:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],
 "agent_scratchpad": "",
 "stop": [
 "\nObservation:",
 "\n\nObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 2:chain:LLMChain > 3:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca
]
}
[llm/end] [1:chain:AgentExecutor > 2:chain:LLMChain > 3:llm:ChatOpenAI] [6.00s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can combi
 "generation_info": null,
 "message": {
 "content": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can
 "additional_kwargs": {},
 "example": false
 }
 }
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 334,
 "completion_tokens": 176,
 "total_tokens": 510
 },
 "model_name": "gpt-3.5-turbo"
 }
}
[chain/end] [1:chain:AgentExecutor > 2:chain:LLMChain] [6.00s] Exiting Chain run with output:
{
 "text": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can combine the
}
[tool/start] [1:chain:AgentExecutor > 4:tool:Python REPL] Entering Tool run with input:
"""
names = [['Harrison', 'Chase'], ['Lang', 'Chain'], ['Dolly', 'Too'], ['Elle', 'Elem'], ['Geoff', 'Fusion'], ['Trance', 'Former'], ['Jen', '
longest_first = ''
shortest_last = names[0][1]
for name in names:
 if len(name[0]) > len(longest_first):
 longest_first = name[0]

```

```

 if len(name[1]) < len(shortest_last):
 shortest_last = name[1]
 combined_name = longest_first + ' ' + shortest_last
 print(combined_name)
 """

[tool/end] [1:chain:AgentExecutor > 4:tool:Python REPL] [0.583ms] Exiting Tool run with output:
"Harrison Too"
[chain/start] [1:chain:AgentExecutor > 5:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase']],
 "agent_scratchpad": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can",
 "stop": [
 "\nObservation:",
 "\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 5:chain:LLMChain > 6:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca"
]
}
[llm/end] [1:chain:AgentExecutor > 5:chain:LLMChain > 6:llm:ChatOpenAI] [5.55s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "I need to check if the code works for other inputs as well.\nAction: Python REPL\nAction Input:\n```\nnames = [['John', 'D"
 "generation_info": null,
 "message": {
 "content": "I need to check if the code works for other inputs as well.\nAction: Python REPL\nAction Input:\n```\nnames = [['John"
 "additional_kwargs": {},
 "example": false
 }
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 517,
 "completion_tokens": 134,
 "total_tokens": 651
 },
 "model_name": "gpt-3.5-turbo"
 }
]
}
[chain/end] [1:chain:AgentExecutor > 5:chain:LLMChain] [5.55s] Exiting Chain run with output:
{
 "text": "I need to check if the code works for other inputs as well.\nAction: Python REPL\nAction Input:\n```\nnames = [['John', 'Doe'],"
}
[tool/start] [1:chain:AgentExecutor > 7:tool:Python REPL] Entering Tool run with input:
"""
names = [['John', 'Doe'], ['Jane', 'Doe'], ['Max', 'Power'], ['Peter', 'Parker']]
longest_first = ''
shortest_last = names[0][1]
for name in names:
 if len(name[0]) > len(longest_first):
 longest_first = name[0]
 if len(name[1]) < len(shortest_last):
 shortest_last = name[1]
combined_name = longest_first + ' ' + shortest_last
print(combined_name)
"""

[tool/end] [1:chain:AgentExecutor > 7:tool:Python REPL] [0.53ms] Exiting Tool run with output:
"Peter Doe"
[chain/start] [1:chain:AgentExecutor > 8:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase']],
 "agent_scratchpad": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can",
 "stop": [
 "\nObservation:",
 "\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 8:chain:LLMChain > 9:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca"
]
}
[llm/end] [1:chain:AgentExecutor > 8:chain:LLMChain > 9:llm:ChatOpenAI] [6.21s] Exiting LLM run with output:

```





```
[chain/end] [1:chain:AgentExecutor > 11:chain:LLMChain] [4.11s] Exiting Chain run with output:
{
 "text": "I need to check if the code works for a list with only one name.\nAction: Python REPL\nAction Input:\n```\nnames = [['John', 'Do
]
[tool/start] [1:chain:AgentExecutor > 13:tool:Python REPL] Entering Tool run with input:
"""
names = [['John', 'Doe']]
longest_first = ''
shortest_last = names[0][1]
for name in names:
 if len(name[0]) > len(longest_first):
 longest_first = name[0]
 if len(name[1]) < len(shortest_last):
 shortest_last = name[1]
combined_name = longest_first + ' ' + shortest_last
print(combined_name)
"""
[tool/end] [1:chain:AgentExecutor > 13:tool:Python REPL] [0.4029999999999997ms] Exiting Tool run with output:
"John Doe"
[chain/start] [1:chain:AgentExecutor > 14:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],
 "agent_scratchpad": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can
 "stop": [
 "\nObservation:",
 "\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 14:chain:LLMChain > 15:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca
]
}
[llm/end] [1:chain:AgentExecutor > 14:chain:LLMChain > 15:llm:ChatOpenAI] [1.34s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "I now know the final answer\nFinal Answer: Peter Doe",
 "generation_info": null,
 "message": {
 "content": "I now know the final answer\nFinal Answer: Peter Doe",
 "additional_kwargs": {},
 "example": false
 }
]
],
 {
 "token_usage": {
 "prompt_tokens": 952,
 "completion_tokens": 12,
 "total_tokens": 964
 },
 "model_name": "gpt-3.5-turbo"
 }
]
}
[chain/end] [1:chain:AgentExecutor > 14:chain:LLMChain] [1.35s] Exiting Chain run with output:
{
 "text": "I now know the final answer\nFinal Answer: Peter Doe"
}
[chain/end] [1:chain:AgentExecutor] [23.22s] Exiting Chain run with output:
{
 "output": "Peter Doe"
}
[chain/start] [1:chain:AgentExecutor] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],
}
[chain/start] [1:chain:AgentExecutor > 2:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],
 "agent_scratchpad": "",
 "stop": [
 "\nObservation:",
 "\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 2:chain:LLMChain > 3:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
```

```

 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca
]
}
[llm/end] [1:chain:AgentExecutor > 2:chain:LLMChain > 3:llm:ChatOpenAI] [6.00s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can combi
 "generation_info": null,
 "message": {
 "content": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can
 "additional_kwargs": {},
 "example": false
 }
]
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 334,
 "completion_tokens": 176,
 "total_tokens": 510
 },
 "model_name": "gpt-3.5-turbo"
 }
}
[chain/end] [1:chain:AgentExecutor > 2:chain:LLMChain] [6.00s] Exiting Chain run with output:
{
 "text": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can combine the
}
[tool/start] [1:chain:AgentExecutor > 4:tool:Python REPL] Entering Tool run with input:
"""
names = [['Harrison', 'Chase'], ['Lang', 'Chain'], ['Dolly', 'Too'], ['Elle', 'Elem'], ['Geoff', 'Fusion'], ['Trance', 'Former'], ['Jen', '
longest_first = ''
shortest_last = names[0][1]
for name in names:
 if len(name[0]) > len(longest_first):
 longest_first = name[0]
 if len(name[1]) < len(shortest_last):
 shortest_last = name[1]
combined_name = longest_first + ' ' + shortest_last
print(combined_name)
"""
[tool/end] [1:chain:AgentExecutor > 4:tool:Python REPL] [0.583ms] Exiting Tool run with output:
"Harrison Too"
[chain/start] [1:chain:AgentExecutor > 5:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],
 "agent_scratchpad": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can
 "stop": [
 "\nObservation:",
 "\n\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 5:chain:LLMChain > 6:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca
]
}
[llm/end] [1:chain:AgentExecutor > 5:chain:LLMChain > 6:llm:ChatOpenAI] [5.55s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "I need to check if the code works for other inputs as well.\nAction: Python REPL\nAction Input:\n```\nnames = [['John', 'D
 "generation_info": null,
 "message": {
 "content": "I need to check if the code works for other inputs as well.\nAction: Python REPL\nAction Input:\n```\nnames = [['John
 "additional_kwargs": {},
 "example": false
 }
]
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 517,
 "completion_tokens": 134,
 "total_tokens": 651
 }
 }
}

```

```

 },
 "model_name": "gpt-3.5-turbo"
}
}
[chain/end] [1:chain:AgentExecutor > 5:chain:LLMChain] [5.55s] Exiting Chain run with output:
{
 "text": "I need to check if the code works for other inputs as well.\nAction: Python REPL\nAction Input:\n```\nnames = [['John', 'Doe'],
}
[tool/start] [1:chain:AgentExecutor > 7:tool:Python REPL] Entering Tool run with input:
"""
names = [['John', 'Doe'], ['Jane', 'Doe'], ['Max', 'Power'], ['Peter', 'Parker']]
longest_first = ''
shortest_last = names[0][1]
for name in names:
 if len(name[0]) > len(longest_first):
 longest_first = name[0]
 if len(name[1]) < len(shortest_last):
 shortest_last = name[1]
combined_name = longest_first + ' ' + shortest_last
print(combined_name)
"""
[tool/end] [1:chain:AgentExecutor > 7:tool:Python REPL] [0.53ms] Exiting Tool run with output:
"Peter Doe"
[chain/start] [1:chain:AgentExecutor > 8:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],
 "agent_scratchpad": "I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can
 "stop": [
 "\nObservation:",
 "\n\tObservation:"
]
}
[llm/start] [1:chain:AgentExecutor > 8:chain:LLMChain > 9:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [
 "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca
]
}
[llm/end] [1:chain:AgentExecutor > 8:chain:LLMChain > 9:llm:ChatOpenAI] [6.21s] Exiting LLM run with output:
{
 "generations": [
 [
 {
 "text": "I need to check if the code works for edge cases where there are ties for longest first name or shortest last name.\nActio
 "generation_info": null,
 "message": {
 "content": "I need to check if the code works for edge cases where there are ties for longest first name or shortest last name.\n
 "additional_kwargs": {},
 "example": false
 }
 }
]
],
 "llm_output": {
 "token_usage": {
 "prompt_tokens": 658,
 "completion_tokens": 164,
 "total_tokens": 822
 },
 "model_name": "gpt-3.5-turbo"
 }
}
[chain/end] [1:chain:AgentExecutor > 8:chain:LLMChain] [6.21s] Exiting Chain run with output:
{
 "text": "I need to check if the code works for edge cases where there are ties for longest first name or shortest last name.\nAction: Pyt
}
[tool/start] [1:chain:AgentExecutor > 10:tool:Python REPL] Entering Tool run with input:
"""
names = [['John', 'Doe'], ['Jane', 'Doe'], ['Max', 'Power'], ['Peter', 'Parker'], ['Joe', 'Dirt'], ['Bob', 'Hope'], ['Tom', 'Cruise']]
longest_first = ''
shortest_last = names[0][1]
for name in names:
 if len(name[0]) >= len(longest_first):
 longest_first = name[0]
 if len(name[1]) <= len(shortest_last):
 shortest_last = name[1]
combined_name = longest_first + ' ' + shortest_last
print(combined_name)
"""
[tool/end] [1:chain:AgentExecutor > 10:tool:Python REPL] [0.55ms] Exiting Tool run with output:
"Peter Doe"

```

```
[chain/start] [1:chain:AgentExecutor > 11:chain:LLMChain] Entering Chain run with input:
{
 "input": "find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],\n\"agent_scratchpad\": \"I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can\n\"stop\": [\n \"\nObservation:\",\n \"\n\tObservation:\"]\n}\n}
[llm/start] [1:chain:AgentExecutor > 11:chain:LLMChain > 12:llm:ChatOpenAI] Entering LLM run with input:
{
 "prompts": [\n "Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca\n]\n}\n[llm/end] [1:chain:AgentExecutor > 11:chain:LLMChain > 12:llm:ChatOpenAI] [4.11s] Exiting LLM run with output:
{\n "generations": [\n [\n {\n "text": "I need to check if the code works for a list with only one name.\nAction: Python REPL\nAction Input:\n```\nnames = [['John\n\"generation_info\": null,\n\"message\": {\n \"content\": \"I need to check if the code works for a list with only one name.\nAction: Python REPL\nAction Input:\n```\nnames = [[\n \"additional_kwargs\": {},\n \"example\": false\n }\n]\n],\n \"llm_output\": {\n \"token_usage\": {\n \"prompt_tokens\": 829,\n \"completion_tokens\": 116,\n \"total_tokens\": 945\n },\n \"model_name\": \"gpt-3.5-turbo\"\n }\n}\n}\n[chain/end] [1:chain:AgentExecutor > 11:chain:LLMChain] [4.11s] Exiting Chain run with output:
{\n \"text\": \"I need to check if the code works for a list with only one name.\nAction: Python REPL\nAction Input:\n```\nnames = [['John', 'Do\n}\n[tool/start] [1:chain:AgentExecutor > 13:tool:Python REPL] Entering Tool run with input:
\"```\n\nnames = [['John', 'Doe']]\nlongest_first = ''\nshortest_last = names[0][1]\nfor name in names:\n if len(name[0]) > len(longest_first):\n longest_first = name[0]\n if len(name[1]) < len(shortest_last):\n shortest_last = name[1]\ncombined_name = longest_first + ' ' + shortest_last\nprint(combined_name)\n```\n\n[tool/end] [1:chain:AgentExecutor > 13:tool:Python REPL] [0.40299999999999997ms] Exiting Tool run with output:
\"John Doe\"\n[chain/start] [1:chain:AgentExecutor > 14:chain:LLMChain] Entering Chain run with input:
{\n \"input\": \"find out the longest first name and shortest last name then combine it to a name\nand print the output: [['Harrison', 'Chase'],\n\"agent_scratchpad\": \"I need to loop through the list of names and keep track of the longest first name and shortest last name. Then I can\n\"stop\": [\n \"\nObservation:\",\n \"\n\tObservation:\"]\n}\n}\n[llm/start] [1:chain:AgentExecutor > 14:chain:LLMChain > 15:llm:ChatOpenAI] Entering LLM run with input:
{\n \"prompts\": [\n \"Human: You are an agent designed to write and execute python code to answer questions.\nYou have access to a python REPL, which you ca\n]\n}\n}\n[llm/end] [1:chain:AgentExecutor > 14:chain:LLMChain > 15:llm:ChatOpenAI] [1.34s] Exiting LLM run with output:
{\n \"generations\": [\n [\n {\n \"text\": \"I now know the final answer\nFinal Answer: Peter Doe\",\n \"generation_info\": null,\n \"message\": {\n
```

```

 "content": "I now know the final answer\nFinal Answer: Peter Doe",
 "additional_kwargs": {},
 "example": false
 }
}
],
"llm_output": {
 "token_usage": {
 "prompt_tokens": 952,
 "completion_tokens": 12,
 "total_tokens": 964
 },
 "model_name": "gpt-3.5-turbo"
}
}
[chain/end] [1:chain:AgentExecutor > 14:chain:LLMChain] [1.35s] Exiting Chain run with output:
{
 "text": "I now know the final answer\nFinal Answer: Peter Doe"
}
[chain/end] [1:chain:AgentExecutor] [23.22s] Exiting Chain run with output:
{
 "output": "Peter Doe"
}

```

```

try:
 result = agent("whats the whether in London now?")
except:
 print("exception on external access")

```