Cookbook

In this notebook we'll take a look at a few common types of sequences to create.

PromptTemplate + LLM

A PromptTemplate -> LLM is a core chain that is used in most other larger chains/systems.

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

API Reference:

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

```
model = ChatOpenAI()
```

```
prompt = ChatPromptTemplate.from_template("tell me a joke about {foo}")
```

```
chain = prompt | model
```

```
chain.invoke({"foo": "bears"})
```

```
    AIMessage(content='Why don\'t bears use cell phones? \n\nBecause
they always get terrible "grizzly" reception!', additional_kwargs={},
example=False)
```

Often times we want to attach kwargs to the model that's passed in. Here's a few examples of that:

Attaching Stop Sequences

```
chain = prompt | model.bind(stop=["\n"])
```

```
chain.invoke({"foo": "bears"})
```

```
    AIMessage(content="Why don't bears use cell phones?",
additional_kwargs={}, example=False)
```

Attaching Function Call information

```
functions = [
    {
      "name": "joke",
      "description": "A joke",
      "parameters": {
        "type": "object",
        "properties": {
          "setup": {
            "type": "string",
            "description": "The setup for the joke"
          },
          "punchline": {
            "type": "string",
            "description": "The punchline for the joke"
          }
        },
        "required": ["setup", "punchline"]
      }
    }
  ]
chain = prompt | model.bind(function_call= {"name": "joke"}, functions=
functions)
```

```
chain.invoke({"foo": "bears"}, config={})
```

```
    AIMessage(content='', additional_kwargs={'function_call': {'name':
'joke', 'arguments': '{\n  "setup": "Why don\'t bears wear shoes?",\n
"punchline": "Because they have bear feet!"\n}'}}, example=False)
```

PromptTemplate + LLM + OutputParser

We can also add in an output parser to easily trasform the raw LLM/ChatModel output into a more workable format

```
from langchain.schema.output_parser import StrOutputParser
```

API Reference:

- StrOutputParser from `langchain.schema.output_parser`

```
chain = prompt | model | StrOutputParser()
```

Notice that this now returns a string - a much more workable format for downstream tasks

```
chain.invoke({"foo": "bears"})
```

```
    "Why don't bears wear shoes?\n\nBecause they have bear feet!"
```

Functions Output Parser

When you specify the function to return, you may just want to parse that directly

```
from langchain.output_parsers.openai_functions import
JsonOutputFunctionsParser
chain = (
    prompt
    | model.bind(function_call= {"name": "joke"}, functions= functions)
    | JsonOutputFunctionsParser()
)
```

API Reference:

- JsonOutputFunctionsParser from `langchain.output_parsers.openai_functions`

```
chain.invoke({"foo": "bears"})
```

```
    {'setup': "Why don't bears wear shoes?",
     'punchline': 'Because they have bear feet!'}
```

```
from langchain.output_parsers.openai_functions import
JsonKeyOutputFunctionsParser
chain = (
    prompt
    | model.bind(function_call= {"name": "joke"}, functions= functions)
    | JsonKeyOutputFunctionsParser(key_name="setup")
)
```

API Reference:

- JsonKeyOutputFunctionsParser from `langchain.output_parsers.openai_functions`

```
chain.invoke({"foo": "bears"})
```

```
    "Why don't bears like fast food?"
```

Passthroughs and itemgetter

Often times when constructing a chain you may want to pass along original input variables to future steps in the chain. How exactly you do this depends on what exactly the input is:

- If the original input was a string, then you likely just want to pass along the string. This can be done with `RunnablePassthrough`. For an example of this, see `LLMChain + Retriever`
- If the original input was a dictionary, then you likely want to pass along specific keys. This can be done with `itemgetter`. For an example of this see `Multiple LLM Chains`

```
from langchain.schema.runnable import RunnablePassthrough
from operator import itemgetter
```

API Reference:

- RunnablePassthrough from `langchain.schema.runnable`

LLMChain + Retriever

Let's now look at adding in a retrieval step, which adds up to a "retrieval-augmented generation" chain

```
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema.runnable import RunnablePassthrough
```

API Reference:

- Chroma from `langchain.vectorstores`
- OpenAIEmbeddings from `langchain.embeddings`
- RunnablePassthrough from `langchain.schema.runnable`

```
# Create the retriever
vectorstore = Chroma.from_texts(["harrison worked at kensho"],
embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
```

```
template = """Answer the question based only on the following context:
{context}

Question: {question}
"""
prompt = ChatPromptTemplate.from_template(template)
```

```
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

```
chain.invoke("where did harrison work?")
```

```
    Number of requested results 4 is greater than number of elements in index 1, updating n_results = 1





    'Harrison worked at Kensho.'
```

```
template = """Answer the question based only on the following context:
{context}

Question: {question}

Answer in the following language: {language}
"""
prompt = ChatPromptTemplate.from_template(template)

chain = {
    "context": itemgetter("question") | retriever,
    "question": itemgetter("question"),
    "language": itemgetter("language")
} | prompt | model | StrOutputParser()
```

```
chain.invoke({"question": "where did harrison work", "language": "italian"})
```

```
    Number of requested results 4 is greater than number of elements in index 1, updating n_results = 1





    'Harrison ha lavorato a Kensho.'
```

Conversational Retrieval Chain

We can easily add in conversation history. This primarily means adding in chat_message_history

```
from langchain.schema.runnable import RunnableMap
from langchain.schema import format_document
```

API Reference:

- RunnableMap from `langchain.schema.runnable`
- format_document from `langchain.schema`

```
from langchain.prompts.prompt import PromptTemplate

_template = """Given the following conversation and a follow up
question, rephrase the follow up question to be a standalone question,
in its original language.

Chat History:
{chat_history}
Follow Up Input: {question}
Standalone question:"""
CONDENSE_QUESTION_PROMPT = PromptTemplate.from_template(_template)
```

API Reference:

- PromptTemplate from `langchain.prompts.prompt`

```
template = """Answer the question based only on the following context:
{context}

Question: {question}
"""
ANSWER_PROMPT = ChatPromptTemplate.from_template(template)
```

```
DEFAULT_DOCUMENT_PROMPT = PromptTemplate.from_template(template="
{page_content}")
def _combine_documents(docs, document_prompt = DEFAULT_DOCUMENT_PROMPT,
document_separator="\n\n"):
    doc_strings = [format_document(doc, document_prompt) for doc in
docs]
    return document_separator.join(doc_strings)
```

```
from typing import Tuple, List
def _format_chat_history(chat_history: List[Tuple]) -> str:
    buffer = ""
    for dialogue_turn in chat_history:
        human = "Human: " + dialogue_turn[0]
        ai = "Assistant: " + dialogue_turn[1]
        buffer += "\n" + "\n".join([human, ai])
    return buffer
```

```
_inputs = RunnableMap(
    {
        "standalone_question": {
            "question": lambda x: x["question"],
            "chat_history": lambda x:
_format_chat_history(x['chat_history'])
        } | CONDENSE_QUESTION_PROMPT | ChatOpenAI(temperature=0) |
StrOutputParser(),
    }
)
_context = {
    "context": itemgetter("standalone_question") | retriever |
_combine_documents,
    "question": lambda x: x["standalone_question"]
}
conversational_qa_chain = _inputs | _context | ANSWER_PROMPT |
ChatOpenAI()
```

```
conversational_qa_chain.invoke({
    "question": "where did harrison work?",
    "chat_history": [],
})
```

```
    Number of requested results 4 is greater than number of elements in
index 1, updating n_results = 1





    AIMessage(content='Harrison was employed at Kensho.',
additional_kwargs={}, example=False)
```

```
conversational_qa_chain.invoke({
    "question": "where did he work?",
    "chat_history": [("Who wrote this notebook?", "Harrison")],
})
```

```
    Number of requested results 4 is greater than number of elements in
index 1, updating n_results = 1





    AIMessage(content='Harrison worked at Kensho.', additional_kwargs=
{}, example=False)
```

With Memory and returning source documents

This shows how to use memory with the above. For memory, we need to manage that outside at the memory. For returning the retrieved documents, we just need to pass them through all the way.

```
from langchain.memory import ConversationBufferMemory
```

API Reference:

- ConversationBufferMemory from `langchain.memory`

```
memory = ConversationBufferMemory(return_messages=True,
output_key="answer", input_key="question")
```

```
# First we add a step to load memory
# This needs to be a RunnableMap because its the first input
loaded_memory = RunnableMap(
    {
        "question": itemgetter("question"),
        "memory": memory.load_memory_variables,
    }
)
# Next we add a step to expand memory into the variables
expanded_memory = {
    "question": itemgetter("question"),
    "chat_history": lambda x: x["memory"]["history"]
}
```

```
# Now we calculate the standalone question
standalone_question = {
    "standalone_question": {
        "question": lambda x: x["question"],
        "chat_history": lambda x:
_format_chat_history(x['chat_history'])
    } | CONDENSE_QUESTION_PROMPT | ChatOpenAI(temperature=0) |
StrOutputParser(),
}
# Now we retrieve the documents
retrieved_documents = {
    "docs": itemgetter("standalone_question") | retriever,
    "question": lambda x: x["standalone_question"]
}
# Now we construct the inputs for the final prompt
final_inputs = {
    "context": lambda x: _combine_documents(x["docs"]),
    "question": itemgetter("question")
}
# And finally, we do the part that returns the answers
answer = {
    "answer": final_inputs | ANSWER_PROMPT | ChatOpenAI(),
    "docs": itemgetter("docs"),
}
# And now we put it all together!
final_chain = loaded_memory | expanded_memory | standalone_question |
retrieved_documents | answer
```

```
inputs = {"question": "where did harrison work?"}
result = final_chain.invoke(inputs)
result
```

```
    Number of requested results 4 is greater than number of elements in
index 1, updating n_results = 1





    {'answer': AIMessage(content='Harrison was employed at Kensho.',
additional_kwargs={}, example=False),
     'docs': [Document(page_content='harrison worked at kensho',
metadata={})]}
```

```
# Note that the memory does not save automatically
# This will be improved in the future
# For now you need to save it yourself
memory.save_context(inputs, {"answer": result["answer"].content})
```

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

```
    {'history': [HumanMessage(content='where did harrison work?',
additional_kwargs={}, example=False),
      AIMessage(content='Harrison was employed at Kensho.',
additional_kwargs={}, example=False)]}
```

Multiple LLM Chains

This can also be used to string together multiple LLMChains

```
from operator import itemgetter

prompt1 = ChatPromptTemplate.from_template("what is the city {person}
is from?")
prompt2 = ChatPromptTemplate.from_template("what country is the city
{city} in? respond in {language}")

chain1 = prompt1 | model | StrOutputParser()

chain2 = {"city": chain1, "language": itemgetter("language")} | prompt2
| model | StrOutputParser()

chain2.invoke({"person": "obama", "language": "spanish"})
```

```
    'El país en el que nació la ciudad de Honolulu, Hawái, donde nació
Barack Obama, el 44º presidente de los Estados Unidos, es Estados
Unidos.'
```

```
from langchain.schema.runnable import RunnableMap
prompt1 = ChatPromptTemplate.from_template("generate a random color")
prompt2 = ChatPromptTemplate.from_template("what is a fruit of color:
{color}")
```

```
prompt3 = ChatPromptTemplate.from_template("what is countries flag that
has the color: {color}")
prompt4 = ChatPromptTemplate.from_template("What is the color of
{fruit} and {country}")
chain1 = prompt1 | model | StrOutputParser()
chain2 = RunnableMap(steps={"color": chain1}) | {
    "fruit": prompt2 | model | StrOutputParser(),
    "country": prompt3 | model | StrOutputParser(),
} | prompt4
```

API Reference:

- RunnableMap from `langchain.schema.runnable`

```
chain2.invoke({})
```

```
    ChatPromptValue(messages=[HumanMessage(content="What is the color
of A fruit that has a color similar to #7E7DE6 is the Peruvian Apple
Cactus (Cereus repandus). It is a tropical fruit with a vibrant purple
or violet exterior. and The country's flag that has the color #7E7DE6
is North Macedonia.", additional_kwargs={}, example=False)])
```

Router

You can also use the router runnable to conditionally route inputs to different runnables.

```
from langchain.chains import create_tagging_chain_pydantic
from pydantic import BaseModel, Field

class PromptToUse(BaseModel):
    """Used to determine which prompt to use to answer the user's
input."""

    name: str = Field(description="Should be one of `math` or
`english`")
```

API Reference:

- create_tagging_chain_pydantic from `langchain.chains`

```
tagger = create_tagging_chain_pydantic(PromptToUse,
ChatOpenAI(temperature=0))
```

```
chain1 = ChatPromptTemplate.from_template("You are a math genius.
Answer the question: {question}") | ChatOpenAI()
chain2 = ChatPromptTemplate.from_template("You are an english major.
Answer the question: {question}") | ChatOpenAI()
```

```
from langchain.schema.runnable import RouterRunnable
router = RouterRunnable({"math": chain1, "english": chain2})
```

API Reference:

- RouterRunnable from `langchain.schema.runnable`

```
chain = {
    "key": {"input": lambda x: x["question"]} | tagger | (lambda x:
x['text'].name),
    "input": {"question": lambda x: x["question"]}
} | router
```

```
chain.invoke({"question": "whats 2 + 2"})
```

```
    AIMessage(content='Thank you for the compliment! The sum of 2 + 2
is equal to 4.', additional_kwargs={}, example=False)
```

Tools

You can use any LangChain tool easily

```
from langchain.tools import DuckDuckGoSearchRun
```

API Reference:

- DuckDuckGoSearchRun from `langchain.tools`

```
/Users/harrisonchase/.pyenv/versions/3.9.1/envs/langchain/lib/python3.9/si
packages/deeplake/util/check_latest_version.py:32: UserWarning: A newer
version of deeplake (3.6.14) is available. It's recommended that you updat
to the latest version using `pip install -U deeplake`.
      warnings.warn(
```

```
search = DuckDuckGoSearchRun()
```

```
template = """turn the following user input into a search query for a
search engine:

{input}"""
prompt = ChatPromptTemplate.from_template(template)
```

```
chain = prompt | model | StrOutputParser() | search
```

```
chain.invoke({"input": "I'd like to figure out what games are
tonight"})
```

```
    "What sports games are on TV today & tonight? Watch and stream live
sports on TV today, tonight, tomorrow. Today's 2023 sports TV schedule
includes football, basketball, baseball, hockey, motorsports, soccer
and more. Watch on TV or stream online on ESPN, FOX, FS1, CBS, NBC,
ABC, Peacock, Paramount+, fuboTV, local channels and many other
networks. Weather Alerts Alerts Bar. Not all offers available in all
states, please visit BetMGM for the latest promotions for your area.
Must be 21+ to gamble, please wager responsibly. If you or someone ...
Speak of the Devils. Good Morning Arizona. Happy Hour Spots. Jaime's
Local Love. Surprise Squad. Silver Apple. Field Trip Friday. Seen on
TV. Arizona Highways TV. MLB Games Tonight: How to Watch on TV,
Streaming & Odds - Friday, July 28. San Diego Padres' Juan Soto plays
during the first baseball game in a doubleheader, Saturday, July 15,
2023, in Philadelphia. (AP Photo/Matt Slocum) (APMedia) Today's MLB
schedule features top teams in action. Among those games is the Texas
Rangers playing the San Diego ... TV. Cleveland at Chi. White Sox.
1:10pm. Bally Sports. NBCS-CHI. Cleveland Guardians (50-51) are second
place in AL Central and Chicago White Sox (41-61) are fourth place in
```

```
AL Central. The Guardians are 23-27 on the road this season and White
Sox are 21-26 at home. Chi. Cubs at St. Louis."
```

Arbitrary Functions

You can use arbitrary functions in the pipeline

Note that all inputs to these functions need to be a SINGLE argument. If you have a function that accepts multiple arguments, you should write a wrapper that accepts a single input and unpacks it into multiple argument.

```
from langchain.schema.runnable import RunnableLambda

def length_function(text):
    return len(text)

def _multiple_length_function(text1, text2):
    return len(text1) * len(text2)

def multiple_length_function(_dict):
    return _multiple_length_function(_dict["text1"], _dict["text2"])

prompt = ChatPromptTemplate.from_template("what is {a} + {b}")

chain1 = prompt | model

chain = {
    "a": itemgetter("foo") | RunnableLambda(length_function),
    "b": {"text1": itemgetter("foo"), "text2": itemgetter("bar")} |
RunnableLambda(multiple_length_function)
} | prompt | model
```

API Reference:

- RunnableLambda from `langchain.schema.runnable`

```
chain.invoke({"foo": "bar", "bar": "gah"})
```

```
    AIMessage(content='3 + 9 is equal to 12.', additional_kwargs={},
example=False)
```

SQL Database

We can also try to replicate our SQLDatabaseChain using this style.

```
template = """Based on the table schema below, write a SQL query that
would answer the user's question:
{schema}

Question: {question}"""
prompt = ChatPromptTemplate.from_template(template)
```

```
from langchain.utilities import SQLDatabase
```

API Reference:

- SQLDatabase from `langchain.utilities`

```
db = SQLDatabase.from_uri("sqlite:///../../../../notebooks/Chinook.db")
```

```
def get_schema(_):
    return db.get_table_info()
```

```
def run_query(query):
    return db.run(query)
```

```
inputs = {
    "schema": RunnableLambda(get_schema),
    "question": itemgetter("question")
}
sql_response = (
        RunnableMap(inputs)
        | prompt
        | model.bind(stop=["\nSQLResult:"])
        | StrOutputParser()
    )
```

```
sql_response.invoke({"question": "How many employees are there?"})
```

```
    'SELECT COUNT(*) \nFROM Employee;'
```

```
template = """Based on the table schema below, question, sql query, and
sql response, write a natural language response:
{schema}

Question: {question}
SQL Query: {query}
SQL Response: {response}"""
prompt_response = ChatPromptTemplate.from_template(template)
```

```
full_chain = (
    RunnableMap({
        "question": itemgetter("question"),
        "query": sql_response,
    })
    | {
        "schema": RunnableLambda(get_schema),
        "question": itemgetter("question"),
        "query": itemgetter("query"),
        "response": lambda x: db.run(x["query"])
    }
    | prompt_response
    | model
)
```

```
full_chain.invoke({"question": "How many employees are there?"})
```

```
    AIMessage(content='There are 8 employees.', additional_kwargs={},
example=False)
```

Code Writing

```
from langchain.utilities import PythonREPL
from langchain.prompts import SystemMessagePromptTemplate,
HumanMessagePromptTemplate
```

API Reference:

- PythonREPL from `langchain.utilities`
- SystemMessagePromptTemplate from `langchain.prompts`
- HumanMessagePromptTemplate from `langchain.prompts`

~~~
template = """Write some python code to solve the user's problem.

Return only python code in Markdown format, eg:

```python
....
```"""
prompt = ChatPromptTemplate(messages=[
    SystemMessagePromptTemplate.from_template(template),
    HumanMessagePromptTemplate.from_template("{input}")
])
~~~

````
def _sanitize_output(text: str):
    _, after = text.split("```python")
    return after.split("```")[0]
````

```
chain = prompt | model | StrOutputParser() | _sanitize_output |
PythonREPL().run
```

```
chain.invoke({"input": "whats 2 plus 2"})
```

```
    Python REPL can execute arbitrary code. Use with caution.





    '4\n'
```

Memory

This shows how to add memory to an arbitrary chain. Right now, you can use the memory classes but need to hook it up manually

```
from langchain.memory import ConversationBufferMemory
from langchain.schema.runnable import RunnableMap
from langchain.prompts import MessagesPlaceholder
model = ChatOpenAI()
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful chatbot"),
    MessagesPlaceholder(variable_name="history"),
    ("human", "{input}")
])
```

API Reference:

- ConversationBufferMemory from `langchain.memory`
- RunnableMap from `langchain.schema.runnable`
- MessagesPlaceholder from `langchain.prompts`

```
memory = ConversationBufferMemory(return_messages=True)
```

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

```
    {'history': []}
```

```
chain = RunnableMap({
    "input": lambda x: x["input"],
    "memory": memory.load_memory_variables
}) | {
    "input": lambda x: x["input"],
    "history": lambda x: x["memory"]["history"]
} | prompt | model
```

```
inputs = {"input": "hi im bob"}
response = chain.invoke(inputs)
```

```
response
```

```
    AIMessage(content='Hello Bob! How can I assist you today?', additional_kwargs={}, example=False)
```

```
memory.save_context(inputs, {"output": response.content})
```

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

```
    {'history': [HumanMessage(content='hi im bob', additional_kwargs={}, example=False),
      AIMessage(content='Hello Bob! How can I assist you today?', additional_kwargs={}, example=False)]}
```

```
inputs = {"input": "whats my name"}
response = chain.invoke(inputs)
response
```

```
    AIMessage(content='Your name is Bob. You mentioned it in your previous message. Is there anything else I can help you with, Bob?', additional_kwargs={}, example=False)
```

Moderation

This shows how to add in moderation (or other safeguards) around your LLM application.

```
from langchain.chains import OpenAIModerationChain
from langchain.llms import OpenAI
```

API Reference:

- OpenAIModerationChain from `langchain.chains`
- OpenAI from `langchain.llms`

```
moderate = OpenAIModerationChain()
```

```
model = OpenAI()
prompt = ChatPromptTemplate.from_messages([
    ("system", "repeat after me: {input}")
])
```

```
chain = prompt | model
```

```
chain.invoke({"input": "you are stupid"})
```

```
    '\n\nYou are stupid.'
```

```
moderated_chain = chain | moderate
```

```
moderated_chain.invoke({"input": "you are stupid"})
```

```
    {'input': '\n\nYou are stupid.',
     'output': "Text was found that violates OpenAI's content policy."}
```