LangChain Expression Language (LCEL)

In [1]:
```
import os
import openai

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file
openai.api_key = os.environ['OPENAI_API_KEY']
```

In [2]:
```
#!pip install pydantic==1.10.8
```

In [3]:
```
from langchain.prompts import ChatPromptTemplate
from langchain.chat_models import ChatOpenAI
#this specific output parser will take ouput and will convert it to string
from langchain.schema.output_parser import StrOutputParser
```

Simple Chain

In [4]:
```
#Making Prompt Template
prompt = ChatPromptTemplate.from_template(
    "tell me a short joke about {topic}"
)
model = ChatOpenAI()
output_parser = StrOutputParser()
```

In [5]:
```
#Making a simple chain that will pass Prompt to model and output of model to c
chain = prompt | model | output_parser
```

In [6]:
```
#Now to call the chain we will use invoke() method and we will give input of t

#**invoke() method calls the runnable using only 1 input:**


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

Out[6]:
```
"Why don't bears wear shoes?\n\nBecause they have bear feet!"
```

More complex chain

And Runnable Map to supply user-provided inputs to the prompt.

In [7]:
```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import DocArrayInMemorySearch
```

In [8]:
```
#stroing data in the vector database
vectorstore = DocArrayInMemorySearch.from_texts(
    ["harrison worked at kensho", "bears like to eat honey"],
    #embeddings that will be used to convert text
    embedding=OpenAIEmbeddings()
)
retriever = vectorstore.as_retriever()
```

In [9]:
```
#use get_relevant_documents() method to get the inforamtion from db retriever
retriever.get_relevant_documents("where did harrison work?")
```

Out[9]:
```
[Document(page_content='harrison worked at kensho'),
 Document(page_content='bears like to eat honey')]
```

In [10]:
```
retriever.get_relevant_documents("what do bears like to eat")
```

Out[10]:
```
[Document(page_content='bears like to eat honey'),
 Document(page_content='harrison worked at kensho')]
```

In [11]:
```
template = """Answer the question based only on the following context:
{context}

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

In [12]:
```
from langchain.schema.runnable import RunnableMap
```

I. Firstly we will take user input, then we will take context and then will pass both to Prompt Template.

II. Pass the Prompt to the Model.

III. Pass the output of model to Output Parser (convert chat message to string).

So firstly we need something that takes Single input (in this case question) and turns it dictionary (by adding context from db too), so for this reason we will use **RunnableMap()**.

In [13]:
```
chain = RunnableMap({
    "context": lambda x: retriever.get_relevant_documents(x["question"]),
    "question": lambda x: x["question"]
}) | prompt | model | output_parser
```

In [14]:
```
chain.invoke({"question": "where did harrison work?"})
```

Out[14]:
```
'Harrison worked at Kensho.'
```

In [15]:
```
inputs = RunnableMap({
    "context": lambda x: retriever.get_relevant_documents(x["question"]),
    "question": lambda x: x["question"]
})
```

In [16]:
```
inputs.invoke({"question": "where did harrison work?"})
```

Out[16]:
```
{'context': [Document(page_content='harrison worked at kensho'),
  Document(page_content='bears like to eat honey')],
 'question': 'where did harrison work?'}
```

Bind

We can use Bind to attach parameters to model/runnables.

Now we will use OpenAI Functions to attach them with models using Bind.

In [17]:
```
functions = [
    {
      "name": "weather_search",
      "description": "Search for weather given an airport code",
      "parameters": {
        "type": "object",
        "properties": {
          "airport_code": {
            "type": "string",
            "description": "The airport code to get the weather for"
          },
        },
        "required": ["airport_code"]
      }
    }
  ]
```

In [18]:
```
prompt = ChatPromptTemplate.from_messages(
    [
        ("human", "{input}")
    ]
)

#binding the function
model = ChatOpenAI(temperature=0).bind(functions=functions)
```

In [19]:
```
runnable = prompt | model
```

In [20]:
```
runnable.invoke({"input": "what is the weather in sf"})
```

Out[20]:
```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'weather_
search', 'arguments': '{\n  "airport_code": "SFO"\n}'}})
```

We can have more than one functions:

In [22]:
```
functions = [
    {
      "name": "weather_search",
      "description": "Search for weather given an airport code",
      "parameters": {
        "type": "object",
        "properties": {
          "airport_code": {
            "type": "string",
            "description": "The airport code to get the weather for"
          },
        },
        "required": ["airport_code"]
      }
    },
        {
      "name": "sports_search",
      "description": "Search for news of recent sport events",
      "parameters": {
        "type": "object",
        "properties": {
          "team_name": {
            "type": "string",
            "description": "The sports team to search for"
          },
        },
        "required": ["team_name"]
      }
    }
  ]
```

In [23]:
```
model = model.bind(functions=functions)
```

In [24]:
```
runnable = prompt | model
```

In [25]:
```
runnable.invoke({"input": "how did the patriots do yesterday?"})
```

Out[25]:
```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'sports_s
earch', 'arguments': '{\n  "team_name": "patriots"\n}'}})
```

Fallbacks

One of powerful features of LCEL is that you can attach Fallbacks with entire sequences.

In [26]:
```
from langchain.llms import OpenAI
import json
```

We will use very early version of openai model, and we will convert the output to json, It is possible that this older version do not let the output converted to json. So chain will break:

In [27]:
```
simple_model = OpenAI(
    temperature=0,
    max_tokens=1000,
    model="text-davinci-001"
)
simple_chain = simple_model | json.loads
```

In [28]:
```
challenge = "write three poems in a json blob, where each poem is a json blob
```

Using model to get simple ouput:

In [29]:
```
simple_model.invoke(challenge)
```

Out[29]:
```
'\n\n["The Waste Land","T.S. Eliot","April is the cruelest month, breeding l
ilacs out of the dead land"]\n\n["The Raven","Edgar Allan Poe","Once upon a
midnight dreary, while I pondered, weak and weary"]\n\n["Ode to a Nightingal
e","John Keats","Thou still unravish\'d bride of quietness, Thou foster-chil
d of silence and slow time"]'
```

Note: The next line is expected to fail.

In [ ]:
```
simple_chain.invoke(challenge)
```

In [32]:
```
model = ChatOpenAI(temperature=0)
new_chain = model | StrOutputParser() | json.loads
```

In [33]:
```
new_chain.invoke(challenge)
```

Out[33]:
```
{'poem1': {'title': 'Whispers of the Wind',
  'author': 'Emily Rivers',
  'first_line': 'Softly it comes, the whisper of the wind'},
 'poem2': {'title': 'Silent Serenade',
  'author': 'Jacob Moore',
  'first_line': 'In the stillness of night, a silent serenade'},
 'poem3': {'title': 'Dancing Shadows',
  'author': 'Sophia Anderson',
  'first_line': 'Shadows dance upon the moonlit floor'}}
```

Now we can make Final Chain with Simple Chain (Chain with older version of openai model), Fallback, and Chain with newwer version of openai model.

```
-So for the Final Chain it will execute input with Simple Chain firs
t.
-If it gets the error then it will fall back to new chain
```

We can add more elements/chains in list of fallbacks, so that it should thorugh those in case of simple chain failure.

In [34]:
```
final_chain = simple_chain.with_fallbacks([chain])
```

In [35]:
```
final_chain.invoke(challenge)
```

Out[35]:
```
{'poem1': {'title': 'Whispers of the Wind',
  'author': 'Emily Rivers',
  'first_line': "Softly it blows, the wind's gentle touch"},
 'poem2': {'title': 'Silent Serenade',
  'author': 'Jacob Stone',
  'first_line': 'In moonlit night, a song unheard'},
 'poem3': {'title': 'Dancing Shadows',
  'author': 'Sophia Reed',
  'first_line': 'Shadows sway, a graceful ballet'}}
```

Interface

In [36]:
```
prompt = ChatPromptTemplate.from_template(
    "Tell me a short joke about {topic}"
)
model = ChatOpenAI()
output_parser = StrOutputParser()

chain = prompt | model | output_parser
```

as invoke() calls the runnable/model on single input

In [37]:
```
chain.invoke({"topic": "bears"})
```

Out[37]:
```
"Why don't bears wear shoes?\n\nBecause they already have bear feet!"
```

We can use batch() to call model for multiple inputs:

In [38]:
```
chain.batch([{"topic": "bears"}, {"topic": "frogs"}])
```

Out[38]:
```
["Why don't bears wear shoes?\n\nBecause they already have bear feet!",
 'Why don\'t frogs make good lawyers?\n\nBecause they always "croak" under p
ressure!']
```

stream() returns iteratable and we can loop to get the result from it, useful when LLM takes time and you stream the result in form of words and show it user

In [ ]:
```
for t in chain.stream({"topic": "bears"}):
    print(t)
```

Asynchronous Invoke():

In [39]:
```
response = await chain.ainvoke({"topic": "bears"})
response
```

Out[39]:
```
"Why don't bears wear shoes? \n\nBecause they have bear feet!"
```

In [ ]:
```

```