**LangChain Expression Language**

What is LangChain Expression Language(LCEL) ?

A “minimalist” code layer for creating chains of LangChain components is made possible by the LangChain Expression Language (LCEL), which is an abstraction of some intriguing Python ideas. It basically uses the pipe operator which is similar to Unix commands where we can pass output of previous function to next function using pipe operator.

LCEL comes with strong support for:

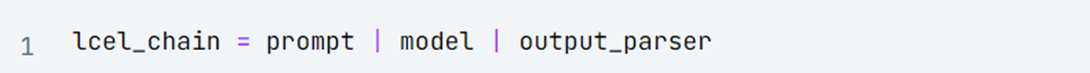
* Superfast development of chains.
* Advanced features such as streaming, async, parallel execution, and more.
* Easy integration with LangSmith and LangServe.

LCEL Syntax

Using  LCEL we create our chain differently using pipe operators (|) rather than Chains objects.

Let us first refresh some concepts related to [LLM](https://www.analyticsvidhya.com/blog/2023/03/an-introduction-to-large-language-models-llms/) chain creation . A basic LLM Chain consists of following 3 components there can be many variations into this which we will learn later in code examples.

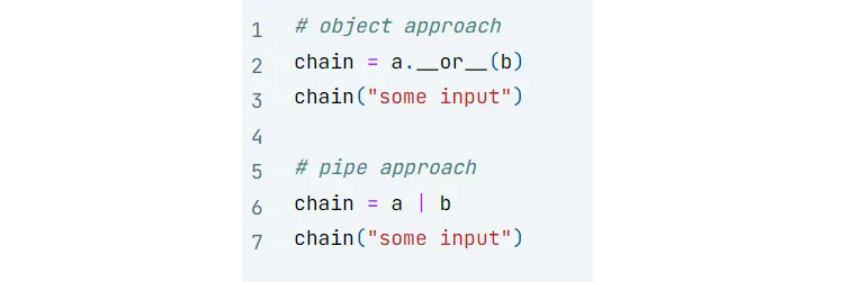
* **LLM**: An abstraction over the paradigm used in Langchain to create completions like Claude, OpenAI GPT3.5, and so on.
* **Prompt**: The LLM object uses this as its input to provide inquiries to the LLM and specify its goals. It is basically a string template which we define with certain placeholders for our variables.
* **Output Parser :** A parser defines how to extract output from response and display it as final response.
* **Chain**:A chain ties up all the above components. It is a series of calls to an [LLM](https://www.analyticsvidhya.com/blog/2023/03/an-introduction-to-large-language-models-llms/), or any stage in the data processing process.



How the Pipe( | ) Operator Works ?

Let us understand how pipe operator works by creating our own small pipe friendly function.

When the Python interpreter sees the **|**operator between two objects (like a | b) it attempts to feed object a into the \_\_or\_\_ method of object b. That means these patterns are equivalent:



Let us use this pipe operator to create our own Runnable Class. It will consume a function and turn it into a function which can be chained with other functions using | operator.

class Runnable:

def \_\_init\_\_(self, func):

self.func = func

def \_\_or\_\_(self, other):

print('or')

def chained\_func(\*args, \*\*kwargs):

# this is nested function in which we create chain of funtion

#here the other function will consume output on this first function

#upon which we call the or operator first element

return other(self.func(\*args, \*\*kwargs))

print('chained func end')

return Runnable(chained\_func)

def \_\_call\_\_(self, \*args, \*\*kwargs):

return self.func(\*args, \*\*kwargs)

#Let's implement this to take the value 3, add 5

Now let us use  this runnable class to chain 2 functions together one is double and second is add one . The below code chains these 2 functions together on input 5.

def double(x):

return 2 \* x

def add\_one(x):

return x + 1

# wrap the functions with Runnable

runnnable\_double = Runnable(double)

runnable\_add\_one = Runnable(add\_one)

# run them using the object approach

chain = runnnable\_double.\_\_or\_\_(runnable\_add\_one)

chain(5) # should return 11

#chain the runnable functions together

double\_then\_add\_one = runnnable\_double | runnable\_add\_one

#invoke the chainLCEL

result = double\_then\_add\_one(5)

print(result) # Output: 11

Let us understand the working of above code one by one :

Creating Runnable Objects

* Runnable(double): This creates a Runnable object that encapsulates the double function. Let’s call this object runnable\_double.
* Runnable(add\_one): Similarly, this one.

Chaining with the | Operator

**runnable\_double | runnable\_add\_one**: This operation triggers the \_\_or\_\_ magic method (operator method) of runnable\_double.

* Inside \_\_or\_\_, a new function called chained\_func is defined. In this function we do chaining of 2 functions on which or operator has been called. This function takes any arguments (\*args, \*\*kwargs) and does the following:
  + It calls runnable\_double.func(\*args, \*\*kwargs) (which is essentially calling double with the given arguments) and passes the result to runnable\_add\_one.func (which calls add\_one).
  + Finally, it returns the output of add\_one b in the return statement.
* The \_\_or\_\_ method returns a new Runnable object (let’s call it double\_then\_add\_one) that stores this chained\_func. Note this chained function is returned when we use or symbol or call method \_\_or\_\_ on the runnable object func 1 | func 2.

Calling the Chained Runnable Object

**double\_then\_add\_one(5):** This calls the calls the \_\_call\_\_ method of the double\_then\_add\_one object.

* The \_\_call\_\_ method in turn executes the chained\_func with the argument 5.
* As explained in step 2, chained\_func calls double(5) (resulting in 10) and then add\_one(10)
* The final result, 11, is returned and assigned to the variable result.

In essence, the Runnable class and the overloaded | operator provide a mechanism to chain functions together, where the output of one function becomes the input of the next. This can lead to more readable and maintainable code when dealing with a series of function calls.

Simple LLM Chain Using LCEL

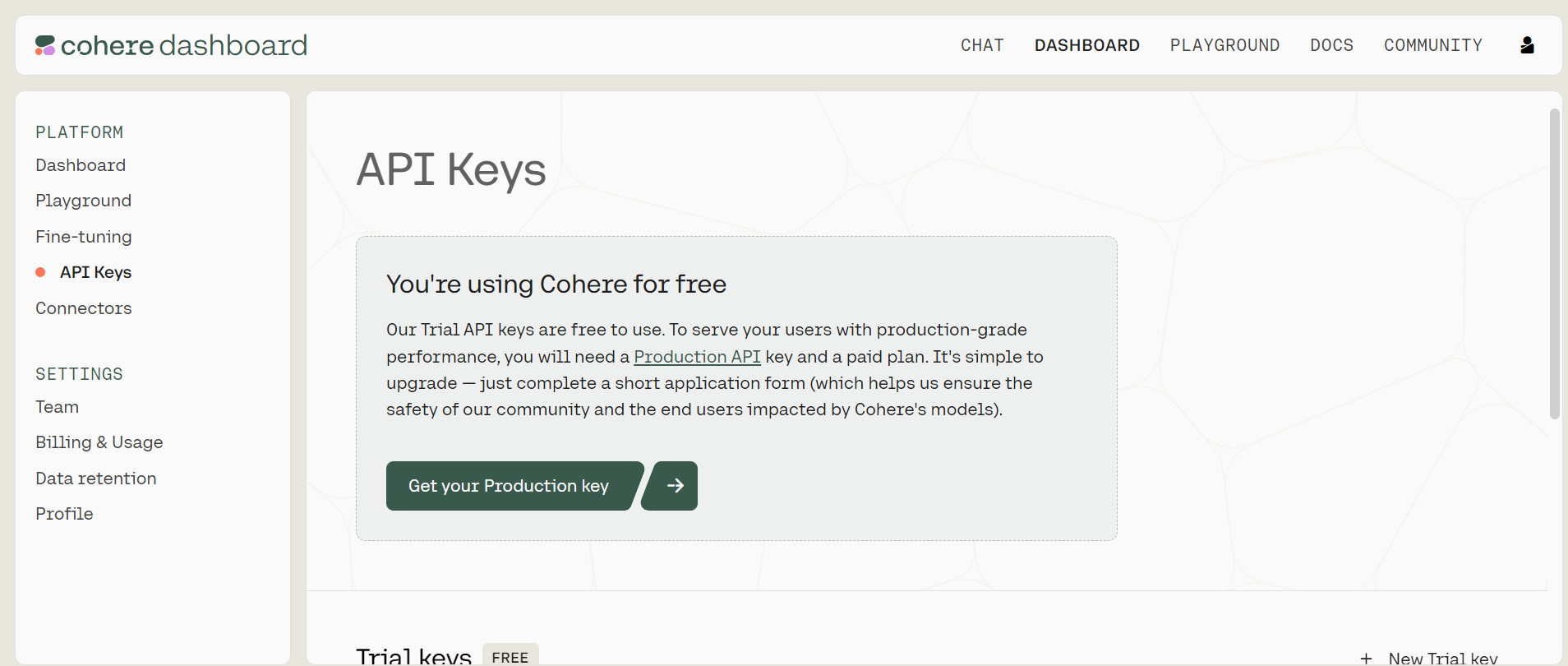
Now we will create a simple LLM chain using LCEL to see how it makes code more readable and intuitive.

# Install Libraries

!pip install langchain\_cohere langchain --quiet

Generate the Cohere API keys

We need to generate the free API key for using Cohere LLM. Visit [website](https://dashboard.cohere.com/welcome/login)  and log in using Google account or github account. Once logged in you will land at a cohere dashboard page as shown below.



Click on API Keys option . You will see a  Trial Free API key is generated.

### Setup Keys

import os

os.environ["COHERE\_API\_KEY"] = "YOUR API KEY"

Create prompt , model , parser and chain

from langchain\_core.prompts import PromptTemplate

from langchain\_core.prompts import ChatPromptTemplate

from langchain\_core.pydantic\_v1 import BaseModel, Field

from langchain\_cohere import ChatCohere

from langchain.schema.output\_parser import StrOutputParser

# LLM Instance

llm = ChatCohere(model="command-r", temperature=0)

#Create Prompt

template = """Question: {question}

Answer: Let's think step by step."""

prompt = PromptTemplate.from\_template(template)

#Create Ouput Parser

output\_parser = StrOutputParser()

# LCEL CHAIN

chain = prompt | llm | output\_parser

question = """

I have five apples. I throw two away. I eat one. How many apples do I have left?

"""

response = chain.invoke({"question": question})

print(response)

Runnables Interface Langchain

When we are working with LCEL we may have the need to modify the flow of values, or the values themselves as they are passed between components — for this, we can use runnables. We can understand how to use Runnables  class provided by Langchain using RAG example.

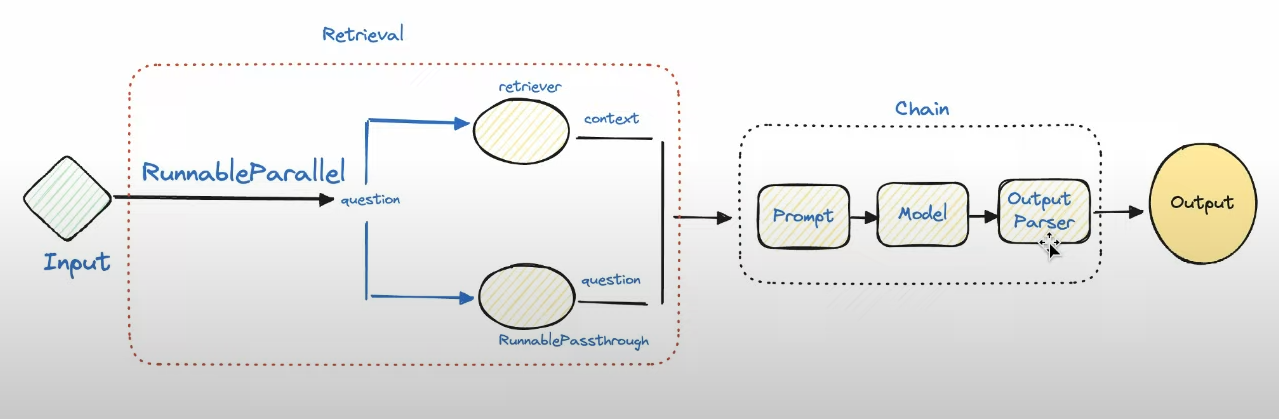
One point about [LangChain Expression Language](https://python.langchain.com/v0.2/docs/concepts/" \l "langchain-expression-language" \t "_blank) is that any two runnables can be “chained” together into sequences. The output of the previous runnable’s .invoke() call is passed as input to the next runnable. This can be done using the pipe operator (|), or the more explicit .pipe() method, which does the same thing.

We shall learn about 3 types of Runnables

* **RunnablePassThrough:** Passes any input as it is to the next component in chain.
* **RunnableParallel:** Passes input to parallel paths simultaneously.
* **RunnableLambda:** Allows to convert any Python function into runnable object which can then be used in chain.

RAG Using Runnable Pass Through and Runnable Parallel

The workflow for the RAG is defined in the image below . Let us now build this RAG to understand usage of Runnable Interfaces.



Installation of Packages

!pip install --quiet langchain langchain\_cohere langchain\_community docarray

Define Vector Stores

We create 2 vector stores to demonstrate the use of Runnable Parallel and Pass through

from langchain.embeddings import CohereEmbeddings

from langchain.vectorstores import DocArrayInMemorySearch

embedding = CohereEmbeddings(

model="embed-english-light-v3.0",

)

vecstore\_a = DocArrayInMemorySearch.from\_texts(

["half the info will be here", "Zoozoo birthday is the 17th September"],

embedding=embedding

)

vecstore\_b = DocArrayInMemorySearch.from\_texts(

["and half here", "Zoozoo was born in 1990"],

embedding=embedding

)

Define Retriever and Chain

Here the input to the “chain.invoke” will be passed to component retrieval where this input is simultaneously passed to two different paths. One is to retriever\_a  whose output is stored in context and passed to next component in chain. The RunnablePassthrough object is used as a “passthrough” take takes any input to the current component (retrieval) and allows us to provide it in the component **output**via the “question” key. Thus input question is available to prompt component in “question” key.

from langchain\_core.runnables import (

RunnableParallel,

RunnablePassthrough

)

retriever\_a = vecstore\_a.as\_retriever()

retriever\_b = vecstore\_b.as\_retriever()

# LLM Instance

llm = ChatCohere(model="command-r", temperature=0)

prompt\_str = """Answer the question below using the context:

Context: {context}

Question: {question}

Answer: """

prompt = ChatPromptTemplate.from\_template(prompt\_str)

retrieval = RunnableParallel(

{"context": retriever\_a, "question": RunnablePassthrough()}

)

chain = retrieval | prompt | llm | output\_parser

Invoke chain

out = chain.invoke("when was Zoozoo born exact year?")

print(out)

**Output:**

Using both retrievers parallelly

We now pass the question to both retrievers parallelly to provide additional context in the prompt.

# Using Both retrievers parallely

prompt\_str = """Answer the question below using the context:

Context:

{context\_a}

{context\_b}

Question: {question}

Answer: """

prompt = ChatPromptTemplate.from\_template(prompt\_str)

retrieval = RunnableParallel(

{

"context\_a": retriever\_a, "context\_b": retriever\_b,

"question": RunnablePassthrough()

}

)

chain = retrieval | prompt | llm | output\_parser

**Output:**

out = chain.invoke("when was Zoozoo born exact date?")

print(out)

Runnable Lambda

Now we will see an example of using runnable Lambda for normal python function similar to what we did earlier in understanding or operator

from langchain\_core.runnables import RunnableLambda

def add\_five(x):

return x + 5

def multiply\_by\_two(x):

return x \* 2

# wrap the functions with RunnableLambda

add\_five = RunnableLambda(add\_five)

multiply\_by\_two = RunnableLambda(multiply\_by\_two)

chain = add\_five | multiply\_by\_two

chain.invoke(3)

Custom Function into Runnable chain

We can use runnable lambda to define our own custom functions and add them into llm chain.

The output of LLM response contains different attributes we will create a custom function extract\_token to display token count for input question and output response

prompt\_str = "You know 1 short line about {topic}?"

prompt = ChatPromptTemplate.from\_template(prompt\_str)

def extract\_token(x):

token\_count = x.additional\_kwargs['token\_count']

response=f'''{x.content} \n Input Token Count: {token\_count['input\_tokens']}

\n Output Token Count:{token\_count['output\_tokens']}'''

return response

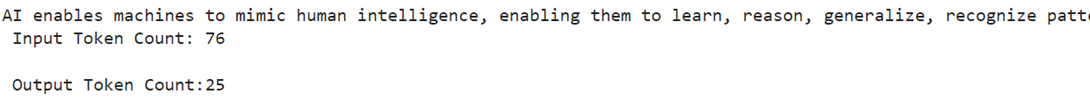
get\_token = RunnableLambda(extract\_token)

chain = prompt | llm | get\_token

**Output:**

output = chain.invoke({"topic": "Artificial Intelligence"})

print(output)



Other Features of LCEL

LCEL has a number of other features also such as async stream batch processing .

* **.invoke()**: The goal is to pass in an input and receive the output—neither more nor less.
* **.batch()**: This is faster than using invoke three times when you wish to supply several inputs to get multiple outputs because it handles the parallelization for you.
* **.stream():** We may begin printing the response before the entire response is complete.

prompt\_str = "You know 1 short line about {topic}?"

prompt = ChatPromptTemplate.from\_template(prompt\_str)

chain = prompt | llm | output\_parser

# ---------invoke--------- #

result\_with\_invoke = chain.invoke("AI")

# ---------batch--------- #

result\_with\_batch = chain.batch(["AI", "LLM", "Vector Database"])

print(result\_with\_batch)

# ---------stream--------- #

for chunk in chain.stream("Artificial Intelligence write 5 lines"):

print(chunk, flush=True, end="")

Async Methods of LCEL

Your application’s frontend and backend are typically independent, which means that requests are made to the backend from the frontend. You may need to manage several requests on your backend at once if you have numerous users.

Since most of the code in LangChain is just waiting between API calls, we can leverage asynchronous code to improve API scalability, if you want to understand why it is important I recommend reading the [concurrent burgers story of the FastAPI documentation.](https://fastapi.tiangolo.com/async/#concurrent-burgers) There is no need to worry about the implementation, because async methods are already available if you use LCEL:

  We can use asynchronous code to increase API scalability because the majority of LangChain’s code consists of basically waiting between API requests. If we use LCEL, async methods are already accessible, thus we don’t need to bother about implementation:

**.ainvoke() / .abatch() / .astream:** asynchronous versions of invoke, batch and stream.

Langchain achieved those “out of the box” features by creating a unified interface called “Runnable”.