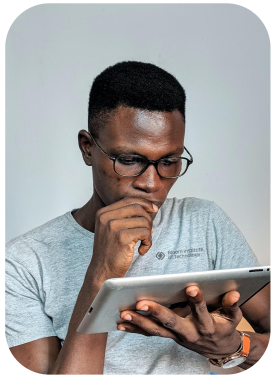
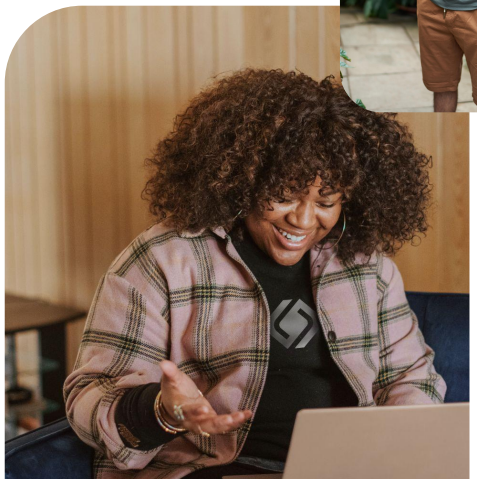
Bloom Institute of Technology

Chaining

In this class, we will discuss chaining and introduce the LangChain Expression Language (LCEL), which helps build chains more quickly and conveniently.




Core Competencies

The student must demonstrate...

1. What is chaining? (10 min)
2. A basic example of using the Lang Chain Expressive Language (10 min)
3. Passing and extracting data with LCEL (20 min)
4. Concurrency in LCEL (10 min)

Chains 101

Chaining involves taking a **series of components and passing data sequentially from one to the next**, following a static path similar to traditional coding. It can include features like asynchronous code and error handling.

	Agents	**Chains**
Core Purpose	Designed for decision-making processes where the LLM decides actions based on observations.	Sequence of tasks to achieve a specific outcome, focusing on structured processes.
Functionality	Involve decision-making and action-taking, iterating through this cycle until task completion; dynamic and adaptive.	Emphasize sequential execution of tasks, often involving data retrieval and processing; efficient for structured workflows.
LangChain Ecosystem	Suitable for complex, flexible scenarios requiring real-time decisions and iteration; LangChain offers various agents for different applications.	Ideal for straightforward, linear workflows with predefined steps; useful for tasks that follow a specific sequence of actions.

Check For Understanding: In which scenarios is it more effective to use chains rather than agents, and how do you decide between the two? Identify 2-3 concrete scenarios for each.

Why Use LCEL

What is LCEL

- The LangChain Expression Language (LCEL) is a language for constructing and managing task chains efficiently.
- Supports features like streaming, **asynchronous processing, and built-in observability.**

Advantages of LCEL Chains

- **Easy Modification:** Modify the internals of a chain by simply adjusting the LCEL.
- **Native Support:** Automatically supports streaming, async, and batch processing.
- **Built-in Observability:** Provides automatic observability at each step.

```python
from langchain_openai import ChatOpenAI
from langchain.prompts.prompt import PromptTemplate
from langchain_core.output_parsers
  import StrOutputParser

# PromptTemplate, ChatOpenAI, and the StrOutputParser
are the components needed for this chain

template = PromptTemplate(template="Tell me the
capital of {country}", input_variables=["country"])
llm = ChatOpenAI()

# Manual way of chaining
prompt = template.format(country="France")
result = llm.predict(prompt)
print(result)

# LCEL way
chain = template | llm | StrOutputParser()
# We added StrOutputParser so we get the content of
the results and not the whole response object

result = chain.invoke({"country": "France"})
print(result)
```

RAG Using LCEL

- LCEL provides built-in support for features like parallel execution, asynchronous processing, and observability, which would be more time-consuming and error-prone to implement manually.
- It simplifies the process of managing complex task chains, ensuring efficient execution and easy modification.

Manual:

```
# Manual way
context = retriever.get_relevant
_documents(prompt)

prompt_with_context = template.invoke({
"query": prompt, "context": context})

results = llm.invoke(prompt_with_context)
```

Access complete RAG code here.

LCEL:

```
# Create a chain that runs tasks in parallel
chain = ( RunnableParallel({"context":
retriever,"query": RunnablePassthrough()})
   # Pass the results to 'template'
   | template
   # Pass the template output to 'llm'
   | llm
   # Parse the LLM output as a string
   | StrOutputParser()
)

results = chain.invoke(prompt)
```

Complex RAG Chain

1. **Setup and Initialization**: Import the necessary libraries and initialize the FAISS vector store with OpenAI embeddings, then create a retriever.

2. **Define the Prompt Template**: Create a chat prompt template with placeholders for context, question, and language.

3. **Construct the Chain**: Define a chain that retrieves the context using the retriever, combines it with the question and language, and then passes through the prompt template, the model, and the output parser.

4. **Invoke the Chain**: Execute the chain with a dictionary containing the question and language, and retrieve the result.

Access LCEL documentation here.

```
from operator import itemgetter
from langchain_community.vectorstores import FAISS
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

vectorstore = FAISS.from_texts(
    ["Byron worked at Rain"], embedding=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()

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 Byron work",
  "language": "italian"})
```

Key LCEL Techniques

1. **Lambdas**: Create a `RunnableParallel` with `RunnablePassthrough` to multiply the input by 3 and add 1, then invoke it with `{"num": 1}`.
2. **Streaming**: Stream a `retrieval_chain` that retrieves context and question via `RunnablePassthrough`, assigns output to `generation_chain`, and print each chunk for the question `where did Byron work?`.

Lambdas:

```
from langchain_core.runnables import
RunnableParallel, RunnablePassthrough

runnable = RunnableParallel(extra=
   RunnablePassthrough.assign(mult=
      lambda x: x["num"] * 3),
      modified=lambda x: x["num"] + 1,
)

runnable.invoke({"num": 1})
```

Streaming:

```
# Plug this code into the complex RAG
example from the previous slide
retrieval_chain = {
   "context": retriever,
   "question": RunnablePassthrough(),
} | RunnablePassthrough.assign(output=
     generation_chain)

stream = retrieval_chain.stream("where
did Byron work?")
for chunk in stream:
   print(chunk)
```

Hands-On Homework

Create a chain that takes a review for a movie and passes it through two different LLMs for sentiment analysis. If they return the same result, then print the answer. If they don't, run a third different LLM to break the tie.

You can use this review for the movie *You've Got Mail* or pick another one if you'd like (try to pick one with some negative and positive words):

```
This film shouldn't work at all. It doesn't have much of a story and the whole dial up internet thing is incredibly dated. However Hanks and Ryan sell it beautifully.
```