Langchain Python

A quick reference guide to LangChain, covering core concepts, LCEL, RAG, agents, and more, to help you build powerful LLM applications.

Core Concepts & Setup

LangChain Basics

What is LangChain?
Framework for building applications powered by large language models (LLMs).

Main Goal:
Chain components together, connect LLMs to data sources, and enable agency for LLMs.

Key Abstraction:
Chains and composability, especially using the LangChain Expression Language (LCEL).

Installation

Basic Installation:
`pip install langchain`

Install Integrations:
`pip install langchain-openai`
`pip install langchain-community`
`pip install langchain-anthropic`
...and so on, for other providers.

Setting up API Keys:
Store API keys as environment variables (recommended).

Example:
`OPENAI_API_KEY=your_openai_api_key`
`export OPENAI_API_KEY`

Models & Prompts

Models (LLMs & Chat Models)

Types:

- `LLM`: Text completion models (e.g., OpenAI's `text-davinci-003`).
- `ChatModel`: Message-based models designed for conversations (e.g., OpenAI's `gpt-3.5-turbo`, `gpt-4`).

Instantiation (Examples):

```
from langchain_openai import OpenAI, ChatOpenAI

llm = OpenAI(model_name="text-davinci-003", temperature=0.7)
chat_model = ChatOpenAI(model_name="gpt-3.5-turbo",
temperature=0.7)
```

```
from langchain_anthropic import Anthropic

model = Anthropic(model_name="claude-3-opus-20240229",
temperature=0.7)
```

Basic Usage:

- `.invoke(text)`: Run the model on a single input.
- `.stream(text)`: Stream the output as it's generated.
- `.batch([text1, text2])`: Run the model on a list of inputs in parallel.

Key Parameters:

- `model_name`: The name of the model to use.
- `temperature`: Controls randomness (0.0 = deterministic, 1.0 = more random).
- `max_tokens`: Limits the length of the output.

Prompts

Prompt Templates:

- `PromptTemplate`: For formatting prompts for LLMs.
- `ChatPromptTemplate`: For formatting prompts for ChatModels (using messages).

Placeholders/Input Variables:
Define input variables in the template and format them with values.

```
from langchain.prompts import PromptTemplate

template = "Tell me a joke about {topic}"
prompt = PromptTemplate.from_template(template)
prompt.format(topic="cats")
```

Message Types (for Chat Models):

- `SystemMessage`: Sets the context or persona for the chatbot.
- `HumanMessage`: Represents user input.
- `AIMessage`: Represents the model's output.

Output Parsers:
Structure the LLM's output into a specific format.

```
from langchain.output_parsers import
CommaSeparatedListOutputParser

output_parser = CommaSeparatedListOutputParser()
template = "List 5 {subject}. {format_instructions}"
prompt = PromptTemplate.from_template(template,
partial_variables={"format_instructions":
output_parser.get_format_instructions()})
chain = prompt | llm | output_parser
chain.invoke({"subject": "colors"})
```

Chains & LCEL

LangChain Expression Language (LCEL)

Concept:
A declarative way to compose chains of components.

Key Operator: The pipe operator `|`.

Example: `prompt | model | output_parser`
This creates a chain that:

1. Formats the prompt.
2. Passes it to the model.
3. Parses the model's output.

RunnablePassthrough:
Passes the input through to the next component.
Useful for adding context or data to the chain.

```
from langchain_core.runnables import RunnablePassthrough

chain = {"topic": RunnablePassthrough()} | prompt | model
chain.invoke("cats")
```

RunnableParallel:
Runs components in parallel.

```
from langchain_core.runnables import RunnableParallel

chain = RunnableParallel({"joke": joke_chain, "fact": fact_chain})
```

Binding Parameters:
Use `.bind()` to fix values for parameters in a component.

```
model = ChatOpenAI().bind(temperature=0.5)
```

Legacy Chains (Briefly)

LLMChain: (Legacy) Combines a PromptTemplate and an LLM.
SequentialChain/SimpleSequentialChain: (Legacy) For running chains in a sequence.

RAG & Data Connection

Vector Stores

Purpose: Store and search embeddings efficiently.

Examples:

- `FAISS`: A fast similarity search library.
- `Chroma`: A popular vector database.
- `Pinecone`: A managed vector database.

Usage:

```
from langchain_community.vectorstores import FAISS

db = FAISS.from_documents(chunks, embeddings)
retriever = db.as_retriever()
results = retriever.get_relevant_documents("my query")
```

Document Loaders

Purpose: Load data from various sources.

Examples:

- `WebBaseLoader`: Loads data from web pages.
- `PyPDFLoader`: Loads data from PDF files.
- `DirectoryLoader`: Loads all files from a directory.
- `CSVLoader`: Loads data from CSV files.

Basic Usage:

```
from langchain_community.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://www.example.com")
documents = loader.load()
```

Text Splitters

Purpose: Break documents into smaller chunks for better LLM performance.

Examples:

- `RecursiveCharacterTextSplitter`: Splits text recursively by characters.
- `CharacterTextSplitter`: Splits text by a single character.

Key Parameters:

- `chunk_size`: The maximum size of each chunk.
- `chunk_overlap`: The amount of overlap between chunks (helps maintain context).

Usage:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
chunks = text_splitter.split_documents(documents)
```

Embeddings

Purpose: Create numerical representations of text.

Examples:

- `OpenAIEmbeddings`: Uses OpenAI's embedding models.
- `HuggingFaceEmbeddings`: Uses Hugging Face's embedding models.

Usage:

```
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
query_result = embeddings.embed_query("hello world")
document_results = embeddings.embed_documents(["hello world"])
```

Building a RAG Chain (LCEL)

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.runnables import RunnableParallel, RunnablePassthrough
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain.chains import create_retrieval_chain

# Load documents
loader = WebBaseLoader("https://docs.smith.langchain.com/overview")
docs = loader.load()

# Split documents
text_splitter = RecursiveCharacterTextSplitter()
splitted_docs = text_splitter.split_documents(docs)

# Create embeddings and vectorstore
embeddings = OpenAIEmbeddings()
db = FAISS.from_documents(splitted_docs, embeddings)

# Create retriever
retriever = db.as_retriever()

# Prompt template
prompt = ChatPromptTemplate.from_messages([("system", """Answer questions based on the context below. If you don't know the answer, just say that you don't know, don't try to make up an answer.

Context: {context}"""), ("user", "{question}")])

# LLM
llm = ChatOpenAI()

# Create a stuff documents chain
document_chain = create_stuff_documents_chain(llm, prompt)

# Create retrieval chain
retrieval_chain = create_retrieval_chain(document_chain, retriever)

# Invoke the chain
response = retrieval_chain.invoke({"question": "how to use langsmith"})
print(response)
```

Agents & Memory

Agents

Concept: LLMs that use tools to interact with the world and make decisions.

Tools: Functions or APIs the agent can use. Defined using the `@tool` decorator or the `Tool` class.

Agent Types:

- `OpenAI Functions Agent`: Uses OpenAI's function calling capability.
- `ReAct`: A reasoning and acting agent.

Agent Executor: The runtime that orchestrates the agent, tools, and model.

```
from langchain_openai import ChatOpenAI
from langchain.agents import create_openai_functions_agent
from langchain.agents import AgentExecutor
from langchain.tools import DuckDuckGoSearchRun

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
tools = [DuckDuckGoSearchRun()]

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant. Use the tools to
best answer the user's questions."),
    ("user", "{input}\n{agent_scratchpad}")
])

agent = create_openai_functions_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True)
agent_executor.invoke({"input": "What is the capital of
France?"})
```

Memory

Concept: Adding state/context to chains or agents to maintain conversation history.

Types:

- `ConversationBufferMemory`: Stores the entire conversation history.
- `ConversationBufferWindowMemory`: Stores a limited window of the conversation history.
- `ConversationSummaryMemory`: Summarizes the conversation history.

Integration (Legacy LLMChain):

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import LLMChain
from langchain_openai import OpenAI
from langchain.prompts import PromptTemplate

prompt_template = """You are a chatbot having a conversation
with a human.\n\n{chat_history}\nHuman:
{human_input}\nChatbot:"""
prompt = PromptTemplate(
    input_variables=["chat_history", "human_input"],
    template=prompt_template
)
memory = ConversationBufferMemory(memory_key="chat_history")
llm_chain = LLMChain(llm=OpenAI(), prompt=prompt, memory=memory)
llm_chain.predict(human_input="Hi, what is the weather today?")
llm_chain.predict(human_input="What about tomorrow?")
```

Callbacks, Debugging, & Use Cases

Callbacks & Debugging

Callbacks: Inject custom logic at different stages of a chain or agent (e.g., `on_llm_start`, `on_chain_end`). Useful for logging, monitoring, or modifying behavior.

Debugging: Set `langchain.debug = True` to enable verbose logging.

LangSmith: A platform for tracing, debugging, and evaluating LangChain applications. Highly recommended for serious development.

Common Use Cases

- Question Answering over Documents (RAG)
- Chatbots
- Summarization
- Data Extraction
- Simple Agents (e.g., calculator, search)