

Build Smarter AI Apps: Empower LLMs with LangChain

Module Cheat Sheet: Introduction to LangChain in GenAI

Package/Method	Description	Code Example
WatsonxLLM	<p>A class from the <code>ibm_watson_machine_learning.foundation_models.extensions.langchain</code> module that creates a LangChain compatible wrapper around IBM's <code>watsonx.ai</code> models.</p>	<pre>from ibm_watsonx_ai.foundation_models import WatsonxLLM from ibm_watson_machine_learning import ModelInference from langchain_core import LangChainCore model_id = 'mistralai/mistral-8x7b-instruct' parameters = { 'max_tokens': 256, 'temperature': 0.2, } credentials = {'url': 'https://us-south.watsonx.ai'} project_id = 'skills-network' model = ModelInference(model_id=model_id, parameters=parameters, credentials=credentials, project_id=project_id) mistral_llm = WatsonxLLM(model=model) response = mistral_llm.invoke("Who is the president of the United States?")</pre>
Message Types	<p>Different types of messages that chat models can use to provide context and control the conversation. The most common message types are <code>SystemMessage</code>, <code>HumanMessage</code>, and <code>AIMessage</code>.</p>	<pre>from langchain_core.messages import HumanMessage, AIMessage, SystemMessage msg = mistral_llm.invoke([SystemMessage(content="You are a helpful assistant"), HumanMessage(content="I enjoy mystery books"), AIMessage(content="What are some good mystery books?")])</pre>
PromptTemplate	<p>A class from the <code>langchain_core.prompts</code> module that helps format prompts with variables. These templates allow you to define a consistent format while leaving placeholders for variables that change with each use case.</p>	<pre>from langchain_core.prompts import PromptTemplate prompt = PromptTemplate.from_template("I like {adjective} {topic}") formatted_prompt = prompt.invoke(input_)</pre>
ChatPromptTemplate	<p>A class from the <code>langchain_core.prompts</code> module that formats a list of chat messages with variables. These templates consist of a list of message templates themselves.</p>	<pre>from langchain_core.prompts import ChatPromptTemplate prompt = ChatPromptTemplate.from_messages([{"role": "system", "content": "You are a helpful assistant"}, {"role": "user", "content": "Tell me a joke about {topic}"]]) input_ = {"topic": "cats"} formatted_messages = prompt.invoke(input_)</pre>

MessagesPlaceholder	<p>A placeholder that allows you to add a list of messages to a specific spot in a ChatPromptTemplate. This capability is useful when you want the user to pass in a list of messages you would slot into a particular spot.</p>	<pre>from langchain_core.prompts import MessagePlaceholder from langchain_core.messages import HumanMessage, AIMessage prompt = ChatPromptTemplate.from_messages([("system", "You are a helpful assistant"), ("user", "What is LangChain?"), ("assistant", "LangChain is a library for building AI applications using large language models."),], messages_placeholder=MessagePlaceholder("msgs")) input_ = {"msgs": [HumanMessage(content="Hello")]} formatted_messages = prompt.invoke(input_)</pre>
JsonOutputParser	<p>A parser that allows users to specify an arbitrary JSON schema and query LLMs for outputs that conform to that schema. A parser is useful for obtaining structured data from LLMs.</p>	<pre>from langchain_core.output_parsers import JsonOutputParser from langchain_core.pydantic_v1 import BaseModel class Joke(BaseModel): setup: str = Field(description="question") punchline: str = Field(description='joke') output_parser = JsonOutputParser(pydantic_type=Joke) format_instructions = output_parser.get_format_instructions() prompt = PromptTemplate(template="Answer the user query.\n\n{{format_instructions}}\n{{input_variables}}\n{{partial_variables}}", input_variables=["query"], partial_variables={"format_instructions": format_instructions}) chain = prompt mixtral_llm output_parser</pre>
CommaSeparatedListOutputParser	<p>A parser used to return a list of comma-separated items. This parser converts the LLM's response into a Python list.</p>	<pre>from langchain.output_parsers import CommaSeparatedListOutputParser output_parser = CommaSeparatedListOutputParser() format_instructions = output_parser.get_format_instructions() prompt = PromptTemplate(template="Answer the user query. {{format_instructions}}\n{{input_variables}}\n{{partial_variables}}", input_variables=["subject"], partial_variables={"format_instructions": format_instructions}) chain = prompt mixtral_llm output_parser result = chain.invoke({"subject": "ice cream"})</pre>
Document	<p>A class from the langchain_core.documents module that contains information about some data. This class has the following two attributes: page_content (the content of the document) and metadata (arbitrary metadata associated with the document).</p>	<pre>from langchain_core.documents import Document doc = Document(page_content="Python is an interpreted programming language. Python's design philosophy prioritizes code readability over performance.", metadata={ 'my_document_id' : 234234, 'my_document_source' : "About Python", 'my_document_create_time' : 1686789 })</pre>

PyPDFLoader	A document loader from the langchain_community.document_loaders that loads PDFs into Document objects. You can use this document loader to extract text content from PDF files.	<pre>from langchain_community.document_loader loader = PyPDFLoader("path/to/document.pdf") documents = loader.load()</pre>
WebBaseLoader	A document loader from the langchain_community.document_loaders that loads content from websites into Document objects. You can use this document loader to extract text content from web pages.	<pre>from langchain_community.document_loader loader = WebBaseLoader("https://python.land") web_data = loader.load()</pre>
CharacterTextSplitter	A text splitter from langchain.text_splitter that splits text into chunks based on characters. This splitter is useful for breaking long documents into smaller, more manageable chunks for processing with LLMs.	<pre>from langchain.text_splitter import CharacterTextSplitter text_splitter = CharacterTextSplitter(chunk_size=200, # Maximum size of each chunk chunk_overlap=20, # Number of characters separator="\n" # Character to split) chunks = text_splitter.split_documents(c)</pre>
RecursiveCharacterTextSplitter	A text splitter from langchain.text_splitter that splits text recursively based on a list of separators. This splitter tries to split on the first separator, then the second separator, and any subsequent separators, until the chunks of text attain the specified size.	<pre>from langchain.text_splitter import RecursiveCharacterTextSplitter text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50, separators=["\n\n", "\n", ". ", " ",],) chunks = text_splitter.split_documents(c)</pre>
WatsonxEmbeddings	A class from langchain_ibm that creates embeddings (vector representations) of text using IBM's watsonx.ai embedding models. You can use these embeddings for semantic search and other vector-based operations.	<pre>from langchain_ibm import WatsonxEmbedding from ibm_watsonx_ai.metanames import EmbedParams embed_params = { EmbedTextParamsMetaNames.TRUNCATE_IN, EmbedTextParamsMetaNames.RETURN_OPT } watsonx_embedding = WatsonxEmbedding(model_id="ibm/slate-125m-english-rtr", url="https://us-south.ml.cloud.ibm.com", project_id="skills-network", params=embed_params,)</pre>

Chroma	A vector store from langchain.vectorstores that stores embeddings and provides methods for similarity search. You can use Chroma for storing and retrieving documents based on semantic similarity.	<pre>from langchain.vectorstores import Chroma // Create a vector store from documents docsearch = Chroma.from_documents(chunks) // Perform a similarity search query = "Langchain" docs = docsearch.similarity_search(query)</pre>
Retrievers	Interfaces that return documents given an unstructured query. Retrievers accept a string query as input and return a list of Document objects as output. You can use vector stores as the backbone of a retriever.	<pre># Convert a vector store to a retriever retriever = docsearch.as_retriever() // Retrieve documents docs = retriever.invoke("Langchain")</pre>
ParentDocumentRetriever	A retriever from langchain.retrievers that splits documents into small chunks for embedding but returns the parent documents during retrieval. This retriever balances accurate embeddings with context preservation.	<pre>from langchain.retrievers import ParentDocumentRetriever from langchain.storage import InMemoryStorage parent_splitter = CharacterTextSplitter() child_splitter = CharacterTextSplitter() vectorstore = Chroma(collection_name="split_parents", embedding_function=watsonx_embedding) store = InMemoryStorage() retriever = ParentDocumentRetriever(vectorstore=vectorstore, docstore=store, child_splitter=child_splitter, parent_splitter=parent_splitter,) retriever.add_documents(documents) retrieved_docs = retriever.invoke("Langchain")</pre>
RetrievalQA	A chain from langchain.chains that answers questions based on retrieved documents. The RetrievalQA chain combines a retriever with an LLM to generate answers based on the retrieved context.	<pre>from langchain.chains import RetrievalQA qa = RetrievalQA.from_chain_type(llm=mixtral_llm, chain_type="stuff", retriever=retriever, return_source_documents=False) query = "what is this paper discussing?" answer = qa.invoke(query)</pre>

ChatMessageHistory	<p>A lightweight wrapper from langchain.memory that provides convenient methods for saving HumanMessages, AIMessages, and then fetching them all. You can use the ChatMessageHistory wrapper to maintain conversation history.</p>	<pre>from langchain.memory import ChatMessageHistory history = ChatMessageHistory() history.add_ai_message("hi!") history.add_user_message("what is the capital of France?") // Access the messages history.messages // Generate a response using the history ai_response = mixtral_llm.invoke(history)</pre>
ConversationBufferMemory	<p>A memory module from langchain.memory that allows for the storage of messages and conversation history. You can use this memory module in conversation chains to maintain context across multiple interactions.</p>	<pre>from langchain.memory import ConversationBufferMemory from langchain.chains import ConversationChain conversation = ConversationChain(llm=mixtral_llm, verbose=True, memory=ConversationBufferMemory()) response = conversation.invoke(input="Hello")</pre>
LLMChain	<p>A basic chain from langchain.chains that combines a prompt template with an LLM. It's the simplest form of chain in LangChain.</p>	<pre>from langchain.chains import LLMChain template = """Your job is to come up with a meal based on {location} YOUR RESPONSE: """ prompt_template = PromptTemplate(template=template) location_chain = LLMChain(llm=mixtral_llm, prompt=prompt_template, output_key='meal') result = location_chain.invoke(input={'location': 'Paris'})</pre>
SequentialChain	<p>A chain from langchain.chains that combines multiple chains in sequence, where the output of one chain becomes the input for the next chain. SequentialChain is useful for multi-step processing.</p>	<pre>from langchain.chains import SequentialChain // First chain - gets a meal based on location location_chain = LLMChain(llm=mixtral_llm, prompt=location_prompt_template, output_key='meal') // Second chain - gets a recipe based on meal dish_chain = LLMChain(llm=mixtral_llm, prompt=dish_prompt_template, output_key='recipe') // Third chain - estimates cooking time recipe_chain = LLMChain(llm=mixtral_llm, prompt=recipe_prompt_template, output_key='time')</pre>

		<pre>// Combine into sequential chain overall_chain = SequentialChain(chains=[location_chain, dish_chain, input_variables=['location'], output_variables=['meal', 'recipe', verbose=True)</pre>
RunnablePassthrough	A component from langchain_core.runnables that allows function chaining to use the 'assign' method, enabling structured multi-step processing.	<pre>from langchain_core.runnables import Runnables // Create each individual chain with the location_chain_lcel = (PromptTemplate.from_template(location_template) mixtral_llm StrOutputParser()) dish_chain_lcel = (PromptTemplate.from_template(dish_template) mixtral_llm StrOutputParser()) time_chain_lcel = (PromptTemplate.from_template(time_template) mixtral_llm StrOutputParser()) overall_chain_lcel = (RunnablePassthrough.assign(meal=dish_chain_lcel) RunnablePassthrough.assign(recipe=time_chain_lcel) RunnablePassthrough.assign(time=location_chain_lcel)) // Run the chain result = overall_chain_lcel.invoke({"location": "Paris"}) pprint(result)</pre>
Tool	A class from langchain_core.tools that represents an interface that an agent, chain, or LLM can use to interact with the world. Tools perform specific tasks like calculations and data retrieval.	<pre>from langchain_core.tools import Tool from langchain_experimental.utilities import PythonREPL python_repl = PythonREPL() python_calculator = Tool(name="Python Calculator", func=python_repl.run, description="Useful for when you need to do some calculations") result = python_calculator.invoke("a = 2 * 3")</pre>
@tool decorator	A decorator from langchain.tools that simplifies the creation of custom tools. This tool automatically converts a function into a Tool object.	<pre>from langchain.tools import tool @tool def search_weather(location: str): """Search for the current weather in {location} # In a real application, this function would query an API return f"The weather in {location} is {weather}"</pre>

create_react_agent	A function from langchain.agents that creates an agent following the ReAct (Reasoning + Acting) framework. This function takes an LLM, a list of tools, and a prompt template as input and returns an agent that can reason and select tools to accomplish tasks.	<pre>from langchain.agents import create_react_agent agent = create_react_agent(llm=mixtral_llm, tools=tools, prompt=prompt)</pre>
AgentExecutor	A class from langchain.agents that manages the execution flow of an agent. This class handles the orchestration between the agent's reasoning and the actual tool execution.	<pre>from langchain.agents import AgentExecutor agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True, handle_parsing_errors=True) result = agent_executor.invoke({"input":</pre>

Author

Hailey Quach



Skills Network