Build Smarter AI Apps: Empower LLMs with LangChain

Module Cheat Sheet: Introduction to LangChain in GenAI

Package/Method	Description	Code Example

WatsonxLLM	A class from the ibm_watson_machine_learning.foundation_models.extensions.langchain module that creates a LangChain compatible wrapper around IBM's watsonx.ai models.	`from ibm_watsonx_ai.foundation_models import ModelInference`
		`from ibm_watson_machine_learning.foundation_models.extensions.langchain import WatsonxLLM`
		
		`model_id = ‘mistralai/mixtral-8x7b-instruct-v01’`
		`parameters = {`
		`    GenParams.MAX_NEW_TOKENS: 256,`
		`    GenParams.TEMPERATURE: 0.2,`
		`}`
		`credentials = {“url”: “https://us-south.ml.cloud.ibm.com”}`
		`project_id = “skills-network”`
		
		`model = ModelInference(`
		`    model_id=model_id,`
		`    params=parameters,`
		`    credentials=credentials,`
		`    project_id=project_id`
		`)`
		
		`mixtral_llm = WatsonxLLM(model=model)`
		`response = mixtral_llm.invoke(“Who is man’s best friend?”)`

Message Types	Different types of messages that chat models can use to provide context and control the conversation. The most common message types are SystemMessage, HumanMessage, and AIMessage.	`from langchain_core.messages import HumanMessage, SystemMessage, AIMessage`
		
		`msg = mixtral_llm.invoke([`
		`    SystemMessage(content=”You are a helpful AI bot that assists a user in choosing the perfect book to read in one short sentence”),`
		`    HumanMessage(content=”I enjoy mystery novels, what should I read?”)`
		`])`

PromptTemplate	A class from the langchain_core.prompts 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.	`from langchain_core.prompts import PromptTemplate`
		
		`prompt = PromptTemplate.from_template(“Tell me one {adjective} joke about {topic}”)`
		`input_ = {“adjective”: “funny”, “topic”: “cats”}`
		
		`formatted_prompt = prompt.invoke(input_)`

ChatPromptTemplate	A class from the langchain_core.prompts module that formats a list of chat messages with variables. These templates consist of a list of message templates themselves.	`from langchain_core.prompts import ChatPromptTemplate`
		
		`prompt = ChatPromptTemplate.from_messages([`
		`    (“system”, “You are a helpful assistant”),`
		`    (“user”, “Tell me a joke about {topic}”)`
		`])`
		
		`input_ = {“topic”: “cats”}`
		`formatted_messages = prompt.invoke(input_)`

MessagesPlaceholder	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.	`from langchain_core.prompts import MessagesPlaceholder`
		`from langchain_core.messages import HumanMessage`
		
		`prompt = ChatPromptTemplate.from_messages([`
		`    (“system”, “You are a helpful assistant”),`
		`    MessagesPlaceholder(“msgs”)`
		`])`
		
		`input_ = {“msgs”: [HumanMessage(content=”What is the day after Tuesday?”)]}`
		`formatted_messages = prompt.invoke(input_)`

JsonOutputParser	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.	`from langchain_core.output_parsers import JsonOutputParser`
		`from langchain_core.pydantic_v1 import BaseModel, Field`
		
		`class Joke(BaseModel):`
		`    setup: str = Field(description=”question to set up a joke”)`
		`    punchline: str = Field(description=”answer to resolve the joke”)`
		
		`output_parser = JsonOutputParser(pydantic_object=Joke)`
		
		`format_instructions = output_parser.get_format_instructions()`
		`prompt = PromptTemplate(`
		`    template=”Answer the user query.\n{format_instructions}\n{query}\n”,`
		`    input_variables=[“query”],`
		`    partial_variables={“format_instructions”: format_instructions},`
		`)`
		
		`chain = prompt | mixtral_llm | output_parser`

CommaSeparatedListOutputParser	A parser used to return a list of comma-separated items. This parser converts the LLM's response into a Python list.	`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}\nList five {subject}.”,`
		`    input_variables=[“subject”],`
		`    partial_variables={“format_instructions”: format_instructions},`
		`)`
		
		`chain = prompt | mixtral_llm | output_parser`
		`result = chain.invoke({“subject”: “ice cream flavors”})`

Document	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).	`from langchain_core.documents import Document`
		
		`doc = Document(`
		`    page_content=”””Python is an interpreted high-level general-purpose programming language.`
		`                    Python’s design philosophy emphasizes code readability with its notable use of significant indentation.”””,`
		`    metadata={`
		`        ‘my_document_id’ : 234234,`
		`        ‘my_document_source’ : “About Python”,`
		`        ‘my_document_create_time’ : 1680013019`
		`    }`
		`)`

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.	`from langchain_community.document_loaders import PyPDFLoader`
		
		`loader = PyPDFLoader(“path/to/document.pdf”)`
		`documents = loader.load()`

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.	`from langchain_community.document_loaders import WebBaseLoader`
		
		`loader = WebBaseLoader(“https://python.langchain.com/v0.2/docs/introduction/”)`
		`web_data = loader.load()`

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.	`from langchain.text_splitter import CharacterTextSplitter`
		
		`text_splitter = CharacterTextSplitter(`
		`    chunk_size=200,  # Maximum size of each chunk`
		`    chunk_overlap=20,  # Number of characters to overlap between chunks`
		`    separator=”\n”  # Character to split on`
		`)`
		`chunks = text_splitter.split_documents(documents)`

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.	`from langchain.text_splitter import RecursiveCharacterTextSplitter`
		
		`text_splitter = RecursiveCharacterTextSplitter(`
		`    chunk_size=500,`
		`    chunk_overlap=50,`
		`    separators=[“\n\n”, “\n”, “. “, “ “, “”]`
		`)`
		`chunks = text_splitter.split_documents(documents)`

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.	`from langchain_ibm import WatsonxEmbeddings`
		`from ibm_watsonx_ai.metanames import EmbedTextParamsMetaNames`
		
		`embed_params = {`
		`    EmbedTextParamsMetaNames.TRUNCATE_INPUT_TOKENS: 3,`
		`    EmbedTextParamsMetaNames.RETURN_OPTIONS: {“input_text”: True},`
		`}`
		
		`watsonx_embedding = WatsonxEmbeddings(`
		`    model_id=”ibm/slate-125m-english-rtrvr”,`
		`    url=”https://us-south.ml.cloud.ibm.com”,`
		`    project_id=”skills-network”,`
		`    params=embed_params,`
		`)`

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.	`from langchain.vectorstores import Chroma`
		
		`// Create a vector store from documents`
		`docsearch = Chroma.from_documents(chunks, watsonx_embedding)`
		
		`// Perform a similarity search`
		`query = “Langchain”`
		`docs = docsearch.similarity_search(query)`

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.	# Convert a vector store to a retriever
		retriever = docsearch.as_retriever()
		
		// Retrieve documents
		docs = retriever.invoke("Langchain")

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.	from langchain.retrievers import ParentDocumentRetriever
		from langchain.storage import InMemoryStore
		
		parent_splitter = CharacterTextSplitter(chunk_size=2000, chunk_overlap=20)
		child_splitter = CharacterTextSplitter(chunk_size=400, chunk_overlap=20)
		
		vectorstore = Chroma(
		    collection_name="split_parents",
		    embedding_function=watsonx_embedding
		)
		
		store = InMemoryStore()
		
		retriever = ParentDocumentRetriever(
		    vectorstore=vectorstore,
		    docstore=store,
		    child_splitter=child_splitter,
		    parent_splitter=parent_splitter,
		)
		
		retriever.add_documents(documents)
		retrieved_docs = retriever.invoke("Langchain")

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.	from langchain.chains import RetrievalQA
		
		qa = RetrievalQA.from_chain_type(
		    llm=mixtral_llm,
		    chain_type="stuff",
		    retriever=docsearch.as_retriever(),
		    return_source_documents=False
		)
		
		query = "what is this paper discussing?"
		answer = qa.invoke(query)

ChatMessageHistory	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.	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.messages)

ConversationBufferMemory	A memory module from langchain.memory that allows for the storage of messages and conversation history. You can use this memory module conversation chains to maintain context across multiple interactions.	from langchain.memory import ConversationBufferMemory
		from langchain.chains import ConversationChain
		
		conversation = ConversationChain(
		    llm=mixtral_llm,
		    verbose=True,
		    memory=ConversationBufferMemory()
		)
		
		response = conversation.invoke(input="Hello, I am a little cat. Who are you?")

LLMChain	A basic chain from langchain.chains that combines a prompt template with an LLM. It's the simplest form of chain in LangChain.	from langchain.chains import LLMChain
		
		template = """Your job is to come up with a classic dish from the area that the users suggests.
		                {location}
		
		            YOUR RESPONSE:
		
		"""
		prompt_template = PromptTemplate(template=template, input_variables=['location'])
		
		location_chain = LLMChain(
		    llm=mixtral_llm,
		    prompt=prompt_template,
		    output_key='meal'
		)
		
		result = location_chain.invoke(input={'location':'China'})

SequentialChain	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.	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'
		)
		
		// Combine into sequential chain
		overall_chain = SequentialChain(
		    chains=[location_chain, dish_chain, recipe_chain],
		    input_variables=['location'],
		    output_variables=['meal', 'recipe', 'time'],
		    verbose=True
		)

RunnablePassthrough	A component from langchain_core.runnables that allows function chaining to use the 'assign' method, enabling structured multi-step processing.	from langchain_core.runnables import RunnablePassthrough
		
		// Create each individual chain with the pipe operator
		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=lambda x: location_chain_lcel.invoke({"location": x["location"]}))
		    | RunnablePassthrough.assign(recipe=lambda x: dish_chain_lcel.invoke({"meal": x["meal"]}))
		    | RunnablePassthrough.assign(time=lambda x: time_chain_lcel.invoke({"recipe": x["recipe"]}))
		)
		
		// Run the chain
		result = overall_chain_lcel.invoke({"location": "China"})
		pprint(result)

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.	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 perform calculations or execute Python code. Input should be valid Python code."
		)
		
		result = python_calculator.invoke("a = 3; b = 1; print(a+b)")

@tool decorator	A decorator from langchain.tools that simplifies the creation of custom tools. This tool automatically converts a function into a Tool object.	from langchain.tools import tool @tool def search_weather(location: str): """Search for the current weather in the specified location.""" # In a real application, this function would call a weather API return f"The weather in {location} is currently sunny and 72°F."

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.	from langchain.agents import create_react_agent
		
		agent = create_react_agent(
		    llm=mixtral_llm,
		    tools=tools,
		    prompt=prompt
		)

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.	from langchain.agents import AgentExecutor
		
		agent_executor = AgentExecutor(
		    agent=agent,
		    tools=tools,
		    verbose=True,
		    handle_parsing_errors=True
		)
		
		result = agent_executor.invoke({"input": "What is the square root of 256?"})

Author

Hailey Quach