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 i
		from ibm_watson_machine_learning.founda
		
		model_id = 'mistralai/mixtral-8x7b-inst
		parameters = {
		    GenParams.MAX_NEW_TOKENS: 256,
		    GenParams.TEMPERATURE: 0.2,
		}
		credentials = {"url": "https://us-south
		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 n```

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 Hum
		
		msg = mixtral_llm.invoke([
		    SystemMessage(content="You are a he
		    HumanMessage(content="I enjoy myste
		])```

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 Prom prompt = PromptTemplate.from_template(' input_ = {"adjective": "funny", "topic' 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 Chat
		
		prompt = ChatPromptTemplate.from_messag
		    ("system", "You are a helpful assis
		    ("user", "Tell me a joke about {top
		])
		
		input_ = {"topic": "cats"}
		formatted_messages = prompt.invoke(inpu```

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 Mess from langchain_core.messages import Hum prompt = ChatPromptTemplate.from_messag ("system", "You are a helpful assis```

|  |  | ```
        MessagesPlaceholder("msgs")
])

input_ = {"msgs": [HumanMessage(content
formatted_messages = prompt.invoke(inpu
``` |
|---|---|---|
| **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 impo
from langchain_core.pydantic_v1 import

class Joke(BaseModel):
    setup: str = Field(description="que
    punchline: str = Field(description=

output_parser = JsonOutputParser(pydant

format_instructions = output_parser.get
prompt = PromptTemplate(
    template="Answer the user query.\n{
    input_variables=["query"],
    partial_variables={"format_instruct
)

chain = prompt | mixtral_llm | output_p
``` |
| **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 Co

output_parser = CommaSeparatedListOutpu

format_instructions = output_parser.get
prompt = PromptTemplate(
    template="Answer the user query. {f
    input_variables=["subject"],
    partial_variables={"format_instruct
)

chain = prompt | mixtral_llm | output_p
result = chain.invoke({"subject": "ice
``` |
| **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 Do

doc = Document(
    page_content="""Python is an interp
                    Python's design phi
    metadata={
        'my_document_id' : 234234,
        'my_document_source' : "About F
        'my_document_create_time' : 168
    }
)
``` |
| **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_loade

loader = PyPDFLoader("path/to/document.
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_loade```
		
		 ```loader = WebBaseLoader("https://python.```
		 ```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 Cha```
		
		 ```text_splitter = CharacterTextSplitter(```
		 ``` chunk_size=200, # Maximum size of```
		 ``` chunk_overlap=20, # Number of char```
		 ``` separator="\n" # Character to spli```
		 ```)```
		 ```chunks = text_splitter.split_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 Rec```
		
		 ```text_splitter = RecursiveCharacterTextS```
		 ``` chunk_size=500,```
		 ``` chunk_overlap=50,```
		 ``` separators=["\n\n", "\n", ". ", " '```
		 ```)```
		 ```chunks = text_splitter.split_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 WatsonxEmbedd```
		 ```from ibm_watsonx_ai.metanames import Em```
		
		 ```embed_params = {```
		 ``` EmbedTextParamsMetaNames.TRUNCATE_I```
		 ``` EmbedTextParamsMetaNames.RETURN_OP1```
		 ```}```
		
		 ```watsonx_embedding = WatsonxEmbeddings(```
		 ``` model_id="ibm/slate-125m-english-rt```
		 ``` url="https://us-south.ml.cloud.ibm.```
		 ``` 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 Chro```
		
		 ```// Create a vector store from documents```
		 ```docsearch = Chroma.from_documents(chunk```
		
		 ```// Perform a similarity search```
		 ```query = "Langchain"```
		 ```docs = docsearch.similarity_search(quer```

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 Parent`
		`from langchain.storage import InMemory`
		
		`parent_splitter = CharacterTextSplitter`
		`child_splitter = CharacterTextSplitter(`
		
		`vectorstore = Chroma(`
		`    collection_name=”split_parents”,`
		`    embedding_function=watsonx_embeddin`
		`)`
		
		`store = InMemoryStore()`
		
		`retriever = ParentDocumentRetriever(`
		`    vectorstore=vectorstore,`
		`    docstore=store,`
		`    child_splitter=child_splitter,`
		`    parent_splitter=parent_splitter,`
		`)`
		
		`retriever.add_documents(documents)`
		`retrieved_docs = retriever.invoke(“Lang`

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 RetrievalQ`
		
		`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 ChatMessag`
		
		`history = ChatMessageHistory()`
		
		`history.add_ai_message(“hi!”)`
		`history.add_user_message(“what is the `
		
		`// Access the messages`
		`history.messages`
		
		`// Generate a response using the histor`
		`ai_response = mixtral_llm.invoke(histor`

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 Conversati`
		`from langchain.chains import Conversati`
		
		`conversation = ConversationChain(`
		`    llm=mixtral_llm,`

| | | ```
verbose=True,
    memory=ConversationBufferMemory()
)

response = conversation.invoke(input="H
``` |
| **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 wi
                {location}

            YOUR RESPONSE:

"""
prompt_template = PromptTemplate(templa

location_chain = LLMChain(
    llm=mixtral_llm,
    prompt=prompt_template,
    output_key='meal'
)

result = location_chain.invoke(input={'
``` |
| **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 Sequential

// First chain - gets a meal based on l
location_chain = LLMChain(
    llm=mixtral_llm,
    prompt=location_prompt_template,
    output_key='meal'
)

// Second chain - gets a recipe based o
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,
    input_variables=['location'],
    output_variables=['meal', 'recipe',
    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 Ru

// Create each individual chain with th
location_chain_lcel = (
    PromptTemplate.from_template(locati
    | mixtral_llm
    | StrOutputParser()
)

dish_chain_lcel = (
    PromptTemplate.from_template(dish_t
    | mixtral_llm
    | StrOutputParser()
)

time_chain_lcel = (
    PromptTemplate.from_template(time_t
    | mixtral_llm
``` |

```
                | StrOutputParser()
)

overall_chain_lcel = (
    RunnablePassthrough.assign(meal=lar
    | RunnablePassthrough.assign(recipe
    | RunnablePassthrough.assign(time=l
)

// Run the chain
result = overall_chain_lcel.invoke({"lo
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.	<pre>from langchain_core.tools import Tool
		from langchain_experimental.utilities i
		
		python_repl = PythonREPL()
		
		python_calculator = Tool(
		    name="Python Calculator",
		    func=python_repl.run,
		    description="Useful for when you ne
		)
		
		result = python_calculator.invoke("a =</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 i
		    # In a real application, this funct
		    return f"The weather in {location}</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_rea
		
		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 AgentExecu
		
		agent_executor = AgentExecutor(
		    agent=agent,
		    tools=tools,
		    verbose=True,
		    handle_parsing_errors=True
		)
		
		result = agent_executor.invoke({"input'</pre>

Author

Hailey Quach