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 im
		from ibm_watson_machine_learning.foundat
		
		model_id = 'mistralai/mixtral-8x7b-instr
		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 ma```

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 Huma
		
		msg = mixtral_llm.invoke([
		    SystemMessage(content="You are a hel
		    HumanMessage(content="I enjoy myster
		])```

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 Promp prompt = PromptTemplate.from_template("T 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 ChatP
		
		prompt = ChatPromptTemplate.from_message
		    ("system", "You are a helpful assist
		    ("user", "Tell me a joke about {topi
		])
		
		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 Messa
from langchain_core.messages import Huma

prompt = ChatPromptTemplate.from_message
    ("system", "You are a helpful assist
    MessagesPlaceholder("msgs")
])

input_ = {"msgs": [HumanMessage(content=
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 impor
from langchain_core.pydantic_v1 import E

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

output_parser = JsonOutputParser(pydanti

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

chain = prompt | mixtral_llm | output_pa
``` |
| **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 Com

output_parser = CommaSeparatedListOutput

format_instructions = output_parser.get_
prompt = PromptTemplate(
    template="Answer the user query. {fc
    input_variables=["subject"],
    partial_variables={"format_instructi
)

chain = prompt | mixtral_llm | output_pa
result = chain.invoke({"subject": "ice c
``` |
| **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 Doc

doc = Document(
    page_content="""Python is an interpr
                    Python's design phil
    metadata={
        'my_document_id' : 234234,
        'my_document_source' : "About Py
        'my_document_create_time' : 1680
    }
)
``` |

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_loader```
		
		 ```loader = PyPDFLoader("path/to/document.p```
		 ```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_loader```
		
		 ```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 Char```
		
		 ```text_splitter = CharacterTextSplitter(```
		 ``` chunk_size=200, # Maximum size of e```
		 ``` chunk_overlap=20, # Number of chara```
		 ``` separator="\n" # Character to split```
		 ```)```
		 ```chunks = text_splitter.split_documents(d```

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 Recu```
		
		 ```text_splitter = RecursiveCharacterTextSp```
		 ``` chunk_size=500,```
		 ``` chunk_overlap=50,```
		 ``` separators=["\n\n", "\n", ". ", " ",```
		 ```)```
		 ```chunks = text_splitter.split_documents(d```

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 WatsonxEmbeddi```
		 ```from ibm_watsonx_ai.metanames import Emb```
		
		 ```embed_params = {```
		 ``` EmbedTextParamsMetaNames.TRUNCATE_IN```
		 ``` EmbedTextParamsMetaNames.RETURN_OPTI```
		 ```}```
		
		 ```watsonx_embedding = WatsonxEmbeddings(```
		 ``` model_id="ibm/slate-125m-english-rtr```
		 ``` url="https://us-south.ml.cloud.ibm.c```
		 ``` 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 Chrom
		
		// Create a vector store from documents
		docsearch = Chroma.from_documents(chunks
		
		// 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 ParentD
		from langchain.storage import InMemorySt
		
		parent_splitter = CharacterTextSplitter(
		child_splitter = CharacterTextSplitter(c
		
		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("Lango
		``

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 ChatMessage\n\nhistory = ChatMessageHistory()\n\nhistory.add_ai_message("hi!")\nhistory.add_user_message("what is the ca\n\n// Access the messages\nhistory.messages\n\n// Generate a response using the history\nai_response = mixtral_llm.invoke(history```
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 Conversatio\nfrom langchain.chains import Conversatio\n\nconversation = ConversationChain(\n llm=mixtral_llm,\n verbose=True,\n memory=ConversationBufferMemory()\n)\n\nresponse = conversation.invoke(input="He```
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\n\ntemplate = """Your job is to come up wit\n {location}\n\n YOUR RESPONSE:\n"""\nprompt_template = PromptTemplate(templat\n\nlocation_chain = LLMChain(\n llm=mixtral_llm,\n prompt=prompt_template,\n output_key='meal'\n)\n\nresult = location_chain.invoke(input={'l```
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 SequentialC\n\n// First chain – gets a meal based on lo\nlocation_chain = LLMChain(\n llm=mixtral_llm,\n prompt=location_prompt_template,\n output_key='meal'\n)\n\n// Second chain – gets a recipe based on\ndish_chain = LLMChain(\n llm=mixtral_llm,\n prompt=dish_prompt_template,\n output_key='recipe'\n)\n\n// Third chain – estimates cooking time\nrecipe_chain = LLMChain(\n llm=mixtral_llm,\n prompt=recipe_prompt_template,\n output_key='time'\n)```

```
// 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 Rur``` ```// Create each individual chain with the``` ```location_chain_lcel = (``` `    PromptTemplate.from_template(locatic` `	mixtral_llm` `	StrOutputParser()` `)` ```dish_chain_lcel = (``` `    PromptTemplate.from_template(dish_te` `	mixtral_llm` `	StrOutputParser()` `)` ```time_chain_lcel = (``` `    PromptTemplate.from_template(time_te` `	mixtral_llm` `	StrOutputParser()` `)` ```overall_chain_lcel = (``` `    RunnablePassthrough.assign(meal=lamb` `	RunnablePassthrough.assign(recipe=` `	RunnablePassthrough.assign(time=la` `)` ```// Run the chain``` ```result = overall_chain_lcel.invoke({"loc``` ```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 in```
		
		```python_repl = PythonREPL()```
		
		```python_calculator = Tool(```
		`    name="Python Calculator",`
		`    func=python_repl.run,`
		`    description="Useful for when you nee`
		`)`
		
		```result = python_calculator.invoke("a = 3```

@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 ir`
		`    # In a real application, this functi`
		`    return f"The weather in {location} i`

| **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_reac

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 AgentExecut

agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True,
    handle_parsing_errors=True
)

result = agent_executor.invoke({"input":
``` |

Author

Hailey Quach