LangChain Ultimate Cheat Sheet

Building LLM Applications with Python (v0.2+)

1 Setup & Models

Installation

```
pip install langchain langchain-openai
```

Chat Models (LLMs) *Interface for models like GPT-4, Claude, Llama.*

```
from langchain_openai import ChatOpenAI

# Initialize Model
model = ChatOpenAI(
    model="gpt-4",
    temperature=0.7, # Creativity (0-1)
    api_key="..."
)

# Basic Invocation
response = model.invoke("Hello world!")
# Output: AIMessage(content="Hi there!")
```

Messages *Standardized message types for chat history.*

```
from langchain_core.messages import (
    HumanMessage, SystemMessage, AIMessage
)

msgs = [
    SystemMessage(content="You are a bot."),
    HumanMessage(content="Hi!")
]
model.invoke(msgs)
```

2 Prompt Templates

Recipes for generating prompts with variable inputs.

String Template

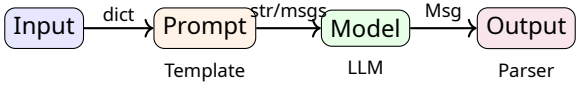
```
from langchain_core.prompts import \
    PromptTemplate

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

Chat Template (Recommended) *Constructs a list of messages.*

```
from langchain_core.prompts import \
    ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a {role}."),
    ("user", "Explain {topic}.")
])

# Usage
prompt.invoke({"role": "teacher",
               "topic": "math"})
```

3 LCEL Syntax

The declarative way to chain components using the pipe '|' operator.



Basic Chain

```
from langchain_core.output_parsers import \
    StrOutputParser

parser = StrOutputParser()

# Chain: Prompt -> Model -> String Parser
chain = prompt | model | parser

# Invoke
result = chain.invoke({"topic": "bears"})
# Result is a string (not AIMessage)
```

Runnable Interface *All LCEL objects support these methods:*

- `.invoke(input)`: Single call.
- `.stream(input)`: Stream chunks.
- `.batch([inputs])`: Parallel processing.

RunnablePassthrough *Passes input unchanged or adds keys.*

```
from langchain_core.runnables import \
    RunnablePassthrough

# x is passed to 'question' AND 'context'
chain = (
    {"context": retriever,
     "question": RunnablePassthrough()}
    | prompt
    | model
)
```

4 Output Parsers

Transform LLM output into structured data.

String Parser

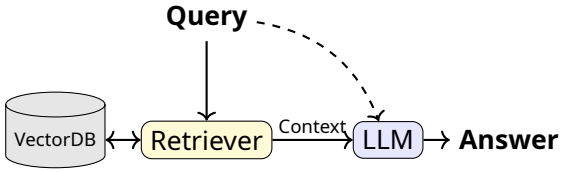
```
# Extracts .content from AIMessage
StrOutputParser()
```

JSON Parser *Enforces and parses JSON output.*

```
from langchain_core.output_parsers import \
    JsonOutputParser
from langchain_core.pydantic_v1 \
    import BaseModel, Field

# Define Schema
class Joke(BaseModel):
    setup: str = Field(description="setup")
    punchline: str = Field(...)

parser = JsonOutputParser(pydantic_object=Joke)

# Inject instructions into prompt
prompt = PromptTemplate(
    template="Joke.\n{format_instructions}",
    input_variables=[],
    partial_variables={
     "format_instructions":
      parser.get_format_instructions()}
)

chain = prompt | model | parser
```

5 RAG (Retrieval)

Retrieval Augmented Generation: Injecting private data.



1. Loading & Splitting

```
from langchain_community.document_loaders \
    import TextLoader
from langchain_text_splitters import \
    RecursiveCharacterTextSplitter

# Load
loader = TextLoader("data.txt")
docs = loader.load()

# Split
splitter = RecursiveCharacterTextSplitter(
```

```
    chunk_size=1000, chunk_overlap=200
)
splits = splitter.split_documents(docs)
```

2. Embedding & Vector Store

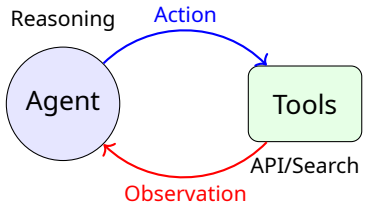
```
from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma

# Embed and Store
vectorstore = Chroma.from_documents(
    documents=splits,
    embedding=OpenAIEmbeddings()
)

# Create Retriever
retriever = vectorstore.as_retriever()
```

3. RAG Chain

```
def format_docs(docs):
    return "\n\n".join(d.page_content for d in docs)

chain = (
    {"context": retriever | format_docs,
     "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

6 Agents & Tools

LLMs that use tools to perform actions.



Defining Tools

```
from langchain_core.tools import tool

@tool
def multiply(a: int, b: int) -> int:
    """Multiplies two numbers."""
    return a * b

tools = [multiply]
```

Tool Calling Agent

```
from langchain.agents import \
    create_tool_calling_agent, AgentExecutor

# Bind tools to model
llm_with_tools = model.bind_tools(tools)

# Create Agent
agent = create_tool_calling_agent(
    llm_with_tools, tools, prompt
)

# Create Executor (Runtime)
agent_executor = AgentExecutor(
    agent=agent, tools=tools, verbose=True
)

agent_executor.invoke({"input": "What is 5*5?"})
```

7 Memory

Adding state to chains.

RunnableWithMessageHistory

```
from langchain_core.chat_history \
    import InMemoryChatMessageHistory
from langchain_core.runnables.history \
    import RunnableWithMessageHistory

store = {} # Key: SessionID, Value: History

def get_session_history(session_id: str):
    if session_id not in store:
        store[session_id] = \
            InMemoryChatMessageHistory()
    return store[session_id]

# Wrap chain
chain_with_history = \
    RunnableWithMessageHistory(
        chain,
        get_session_history
    )

# Invoke with session ID
chain_with_history.invoke(
    {"input": "Hi!"},
    config={"configurable":
            {"session_id": "user1"}}
)
```