Reference Card: LangChain
Yogesh Haribhau Kulkarni

Installation & Imports (LangChain v1)

```
# Core packages (v1.0+)
pip install langchain langchain-core
pip install langchain-groq  # or langchain-openai,
        langchain-anthropic

# For document processing
pip install langchain-community langchain-text-splitters

# For embeddings
pip install langchain-huggingface

# For legacy features (Memory, old chains)
pip install langchain-classic
```

```
# Core Imports (v1 modern syntax)
from langchain_groq import ChatGroq  # Provider-specific imports
from langchain_core.prompts import PromptTemplate,
        ChatPromptTemplate
from langchain_core.messages import HumanMessage, SystemMessage,
        AIMessage
from langchain_core.output_parsers import StrOutputParser,
        PydanticOutputParser

# Agents (NEW in v1)
from langchain.agents import create_agent
from langchain_core.tools import tool

# Memory (moved to langchain-classic)
from langchain_classic.memory import ConversationBufferMemory

# Community connectors
from langchain_community.document_loaders import WebBaseLoader
from langchain_huggingface import HuggingFaceEmbeddings
```

Key Changes in v1:

- `create_agent` replaces `create_react_agent`
- Memory moved to `langchain-classic`
- Provider-specific imports (e.g., `langchain_groq`)

Message Types & Chat

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

messages = [
    SystemMessage(content="You are a helpful AI assistant"),
    HumanMessage(content="What is machine learning?"),
    AIMessage(content="ML is a subset of AI..."),
]

# Chat prompt templates (runnable-friendly)
from langchain_core.prompts import ChatPromptTemplate,
        HumanMessagePromptTemplate

human = HumanMessagePromptTemplate.from_template("{question}")
chat_prompt = ChatPromptTemplate.from_messages([human])

from langchain_groq import ChatGroq
llm = ChatGroq(model="llama-3.3-70b-versatile", temperature=0.0)

# Use pipe operator to run prompt -> llm
chain = chat_prompt | llm
# invoke with a dict containing prompt variables
resp = chain.invoke({"question": "Explain supervised learning in 3
        bullets."})
print(resp)  # raw LLM response
```

Prompt Templates & Output Parsers

```
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser,
        StructuredOutputParser
from pydantic import BaseModel, Field

# Classic PromptTemplate -> used with runnables
prompt = PromptTemplate(
    input_variables=["text","language"],
    template="Translate the following text to {language}:\n\n{text}"
)

# Example of piping prompt -> llm and parsing to string
from langchain_groq import ChatGroq
llm = ChatGroq(model="gemma2-9b-it", temperature=0.2)
chain = prompt | llm | StrOutputParser()
out = chain.invoke({"text": "Hello", "language": "French"})
print(out)  # plain string

# Structured output parser example (schema-backed)
from langchain_core.output_parsers import StructuredOutputParser,
        ResponseSchema
schemas = [
    ResponseSchema(name="summary", description="Short summary"),
    ResponseSchema(name="source", description="source of info")
]
parser = StructuredOutputParser.from_response_schemas(schemas)
prompt2 = PromptTemplate(
    input_variables=["doc"],
    template="Read the document and return json with keys
        'summary' and 'source'.\n\n{doc}\n\n{parser_instructions}"
)
# inject the parser instructions into the prompt
prompt_text = prompt2.format(doc="...",
        parser_instructions=parser.get_format_instructions())
chain2 = PromptTemplate(
    input_variables=["doc"],
    template=prompt_text
) | llm
result = chain2.invoke({"doc": "Long document text"})
# Then parse the result using the parser (or include parser as a
        runnable)
parsed = parser.parse(result)
```

Loaders & Documents (unchanged interfaces)

```
# Text file loader
from langchain_community.document_loaders import TextLoader
loader = TextLoader("file.txt")
docs = loader.load()

# CSV Loader
from langchain_community.document_loaders import CSVLoader
loader = CSVLoader("data.csv")
docs = loader.load()

# Web Page Loader
from langchain_community.document_loaders import WebBaseLoader
loader = WebBaseLoader("https://example.com")
docs = loader.load()

# Directory Loader
from langchain_community.document_loaders import DirectoryLoader
loader = DirectoryLoader("./docs", glob="**/*.txt")
docs = loader.load()

# Unstructured Loader (docx/pdf)
from langchain_community.document_loaders import
        UnstructuredFileLoader
loader = UnstructuredFileLoader("file.docx")
docs = loader.load()
```

Text Splitters & Embeddings

```
from langchain_text_splitters  import CharacterTextSplitter
splitter = CharacterTextSplitter(separator="\n\n", chunk_size=1000,
        chunk_overlap=200)
chunks = splitter.split_text(long_text)

# Example: embeddings -> vectorstore
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS

embeddings =
        HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
vectors = FAISS.from_documents(chunks, embeddings)
```

Runnables: Replacing LLMChain

```
# Legacy:
# chain = LLMChain(llm=llm, prompt=prompt)
# out = chain.run("input")

# Modern (runnable/LCEL) pattern:
from langchain_core.prompts import PromptTemplate
from langchain_groq import ChatGroq


prompt = PromptTemplate(input_variables=["name"], template="Say
        hello to {name}.")
llm = ChatGroq(model="llama-3.3-70b-versatile", temperature=0.0)

# Compose with | (prompt -> llm)
pipeline = prompt | llm
# Invoke: pass a dict of the prompt variables
result = pipeline.invoke({"name": "Alice"})
print(result)
```

Sequential Pipelines (Replacing SimpleSequentialChain)

```
# Legacy: SimpleSequentialChain([...])
# Modern: chain runnables with '|' or RunnableSequence

from langchain_core.runnables import RunnableSequence

p1 = PromptTemplate(input_variables=["text"],
        template="Summarize: {text}")
p2 = PromptTemplate(input_variables=["summary"],
        template="Translate to French: {summary}")

pipeline = (p1 | llm) | (p2 | llm)  # outputs of first feed second
        automatically
# or build an explicit sequence:
sequence = RunnableSequence([p1 | llm, p2 | llm])
out = sequence.invoke({"text": "Long text to summarize"})
print(out)
```

Callbacks & Tracing (same handlers, new manager)

```
from langchain_core.callbacks.base import BaseCallbackHandler
from langchain_core.callbacks.manager import CallbackManager

class MyCallback(BaseCallbackHandler):
    def on_llm_start(self, serialized, prompts, **kwargs):
        print("LLM started:", prompts)
    def on_llm_end(self, response, **kwargs):
        print(”LLM finished”)

cb_manager = CallbackManager([MyCallback()])
llm = ChatOpenAI(callback_manager=cb_manager)
# run as usual with runnables; callbacks will be invoked
```

Local & HF Models (examples kept)

```
# HuggingFace Pipeline (local)
from langchain_community.llms import HuggingFacePipeline
from transformers import AutoTokenizer, AutoModelForCausalLM,
        pipeline

model_id = ”gpt2”
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)
pipe = pipeline(”text−generation”, model=model, tokenizer=tokenizer)
llm_local = HuggingFacePipeline(pipeline=pipe)

# Use as any other llm runnable:
prompt = PromptTemplate(input_variables=[”q”], template=”{q}”)
resp = (prompt | llm_local).invoke({”q”: ”Write a haiku about code.”})
```

Retrieval & RAG (Replacing RetrievalQA)

```
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

# Assume vectorstore is already created
retriever = vectorstore.as_retriever(search_kwargs={”k”: 4})

def format_docs(docs):
    return ”\n\n”.join(doc.page_content for doc in docs)

prompt = ChatPromptTemplate.from_template(”””
Answer based on context:

Context: {context}
Question: {question}
”””)

# Modern RAG chain with LCEL
rag_chain = (
    {
        ”context”: retriever | format_docs,
        ”question”: RunnablePassthrough()
    }
    | prompt
    | llm
    | StrOutputParser()
)

answer = rag_chain.invoke(”What is the capital of France?”)
```

Agents & Tools (v1 Modern Approach)

```
# Modern v1 approach: use create_agent (NOT create_react_agent)
from langchain.agents import create_agent
from langchain_core.tools import tool

# Define tools using @tool decorator
@tool
def calculate(expression: str) −> str:
```

```
    ”””Evaluate a mathematical expression.”””
    try:
        return str(eval(expression))
    except:
        return ”Invalid expression”

@tool
def search_docs(query: str) −> str:
    ”””Search documentation for information.”””
    # Use retriever here if needed
    return f”Results for: {query}”

# Create agent with tools (simple, clean API)
agent = create_agent(
    model=”llama−3.3−70b−versatile”,  # Can use model string or
        ChatGroq object
    tools=[calculate, search_docs],
    system_prompt=”You are a helpful assistant with access to tools.”
)

# Run the agent (returns full message history)
response = agent.invoke({
    ”messages”: [{”role”: ”user”, ”content”: ”What’s 2∗∗10?”}]
})

# Extract final answer
print(response[”messages”][−1].content)
```

v1 Changes:

- `create_agent` replaces `create_react_agent` + `AgentExecutor`
- No more `AgentExecutor` wrapper needed
- Built on LangGraph internally (durable execution)

Memory (Requires langchain-classic in v1)

Important: Memory classes moved to `langchain-classic` in v1.

```
pip install langchain−classic
```

```
# Updated import for v1
from langchain_classic.memory import ConversationBufferMemory

memory = ConversationBufferMemory(memory_key=”chat_history”)

# For new projects, consider:
# − LangGraph’s built−in persistence
# − Custom state management with LCEL
# − Manual message history tracking
```

Note: For production apps, consider using LangGraph for state management instead of legacy memory classes.

LLM Config & Parameters (examples)

```
# temperature / max_tokens unchanged at model construction
from langchain_groq import ChatGroq

llm = ChatGroq(
    model=”llama−3.3−70b−versatile”,
    temperature=0.7,
    max_tokens=500
```

```
)

# streaming & callbacks are supported via callback_manager when
        constructing the model
from langchain_core.callbacks.manager import CallbackManager
llm_streaming = ChatOpenAI(streaming=True,
        callback_manager=cb_manager)

# Using prompt | llm | parser pipeline:
from langchain_core.output_parsers import StrOutputParser
pipeline = PromptTemplate(input_variables=[”q”], template=”{q}”) |
        llm | StrOutputParser()
answer = pipeline.invoke({”q”: ”Write a short definition of RAG.”})
```

Short Examples / Patterns (cheat sheet)

```

from langchain_groq import ChatGroq

llm = ChatGroq(model=”llama−3.3−70b−versatile”, temperature=0)

# Basic prompt −> llm
prompt = PromptTemplate(input_variables=[”x”], template=”Reverse:
        {x}”)
resp = (prompt | llm).invoke({”x”:”abc”})
print(resp)

# Chain with parsing
from langchain_core.output_parsers import PydanticOutputParser
from pydantic import BaseModel
class Person(BaseModel):
    name: str
    age: int

parser = PydanticOutputParser(pydantic_object=Person)
chain = PromptTemplate(input_variables=[”txt”], template=”Return
        json matching Person for: {txt}”) | llm | parser
out = chain.invoke({”txt”: ”Name Bob age 30”})
print(out)
```

V1 Migration Quick Reference

v0 (Old)	v1 (New)
`create_react_agent`	`create_agent`
`AgentExecutor`	Built-in (not needed)
`langchain.memory`	`langchain_classic.memory`
`langchain.schema`	`langchain_core.messages`
`langchain.prompts`	`langchain_core.prompts`
`langchain.document_loaders`	`langchain_community.document_loaders`
`LLMChain`	LCEL with `|` operator
`RetrievalQA`	Custom LCEL RAG chain

Key Requirements:

- Python 3.10+ (Python 3.9 EOL October 2025)
- Install `langchain-classic` for legacy features
- Use provider-specific packages (`langchain-groq`, etc.)