D. Grigorov, *Intermediate Python and Large Language Models*
https://doi.org/10.1007/979-8-8688-1475-4_1

1. LangChain and Python: Basics

Dilyan Grigorov[1]
(1) Varna, Varna, Bulgaria

LangChain is a powerful new framework in Python that simplifies building intelligent applications using natural language processing (NLP) and large language models (LLMs). It reduces complexity, making AI-powered solutions more accessible to developers. At its core, LangChain provides a set of abstractions and utilities that make it easier to build, customize, and deploy NLP-based workflows, such as chatbots, automated data analysis, summarization tools, and much more. Given Python's status as the go-to language for AI and data science, integrating LangChain with Python creates a powerful toolset for developers and data practitioners looking to enhance their NLP projects.

LangChain's primary goal is to simplify how developers interact with language models and manage their outputs in context-rich applications. Typically, when using a language model like OpenAI's GPT, there's a need to set up workflows for input, processing, context handling, and response generation. LangChain provides a framework to define and chain these elements, known as "chains," enabling more complex and sophisticated NLP applications without needing to manually handle all aspects of the process.

Python has a rich set of libraries for machine learning (e.g., TensorFlow, PyTorch) and NLP (e.g., spaCy, NLTK). LangChain seamlessly fits into this ecosystem by offering high-level abstractions that allow developers to quickly integrate language models into their applications. Key benefits include

- **Ease of Integration:** LangChain abstracts much of the complexity involved in setting up prompts, model calls, and response handling, making it easier to build and deploy applications.
- **Modularity and Flexibility:** LangChain enables chaining multiple LLM calls together, combining different models, and adding context to create more advanced applications, such as multistep question-answering systems or conversational agents.
- **Handling Context and Memory:** One of LangChain's strengths is its ability to manage context and memory effectively. For conversational AI or tasks that require understanding of a sequence of interactions, LangChain provides utilities to track and store context throughout the conversation or workflow.
- **Scalability and Deployment:** By working within the Python ecosystem, LangChain can be easily integrated into larger projects, data pipelines, or cloud-deployed applications, making it a practical choice for both experimentation and production-level applications.

With Python and LangChain, developers can build a wide range of NLP applications:

- **Chatbots and Conversational Agents:** Implement agents that can handle context-aware conversations, manage user intents, and respond dynamically to user queries
- **Data Extraction and Summarization:** Create pipelines that process large amounts of text, extract key information, and produce summaries or insights
- **Automated Content Generation:** Use language models to generate content for blogs, reports, or documentation based on given prompts or templates
- **Question-Answering Systems:** Build tools that allow users to ask questions about specific documents or datasets, where the system can pull and present relevant information

In this first chapter, we will explore how to use LangChain with Python to create advanced language model applications, discussing its key components and providing practical examples to get you started.

This chapter also

- Introduces LangChain as a Python framework for building LLM-powered apps like chatbots and summarizers, with a focus on modularity, memory, and context handling
- Covers core concepts: chains, prompts, memory, tools, agents, RAG, data loaders, and integrations
- Explains installation of LangChain and related packages (e.g., langchain-core, langchain-openai, langgraph)
- Provides prompt engineering techniques: role prompting, few-shot, chain prompting, chain-of-thought, alternating messages, and refinement tips
- Describes various chain types: simple, sequential, conversational, multi-input/output, router, control flow, retrieval-aware, and agent chains—with code examples

LangChain Basics and Basic Components

As I mentioned, LangChain is a powerful framework designed for developing applications that integrate large language models (LLMs) like OpenAI's GPT-4 into workflows or pipelines that can perform a variety of complex tasks. It is particularly helpful for creating applications that require language model capabilities, whether for natural language understanding, processing, or generation.

Here are the fundamental components and concepts of LangChain.

Chains

Chains are sequences of operations (or steps) designed to process and transform data. In LangChain, chains can be created to link together multiple steps that involve LLMs, transforming the input through a sequence of transformations or tasks. A simple chain might involve querying an LLM with a prompt, whereas more complex chains can combine multiple actions, like API calls, data retrieval, or conditional logic.

Prompts

Prompts are the input that LLMs use to generate responses. LangChain allows users to design prompts dynamically, enabling the creation of tailored queries based on different scenarios or contexts. You can create

prompt templates that include variables to be filled in based on user inputs or other data.

Memory

Memory allows a chain to retain state throughout a conversation or across multiple interactions. This feature is particularly useful for applications that require context over time, such as chatbots or assistants, where responses need to be informed by the history of the conversation.

Tools and Agents

LangChain provides tools that interact with external systems or APIs, such as databases, search engines, or custom APIs. Agents are advanced chains that can decide which tool to use based on the input they receive. For example, an agent could determine whether to perform a search, fetch data from a database, or generate a response directly.

Retrieval-Augmented Generation (RAG)

RAG is a method where LLMs are combined with external data sources to enhance their outputs. Unlike standard LLM queries that rely on pre-trained knowledge, RAG dynamically retrieves up-to-date information from external sources before responding, ensuring better accuracy and contextual awareness. For example, an LLM may query a knowledge base or a search engine to find relevant information before generating a response. LangChain supports RAG through its retrieval tools and agents, making it suitable for applications that require updated or domain-specific information.

Data Loaders

LangChain includes data loaders for various types of data sources, like local files, APIs, and databases. These loaders help convert raw data into a format that can be processed or queried by an LLM.

Integrations and Extensibility

LangChain is designed to integrate easily with other tools and libraries. It supports various LLM back ends (such as OpenAI, Hugging Face, and others) and can be extended with custom chains, agents, or tools. This makes it flexible for creating custom applications across different domains.

LLM Outputs and Postprocessing

LangChain provides ways to interpret and process the outputs from LLMs. Since LLMs may produce complex or unstructured data, LangChain includes components for parsing, formatting, and further transforming these outputs to be more usable for the application.

By leveraging these concepts, LangChain allows for building powerful, customizable LLM-powered applications efficiently.

LangChain Installation

The LangChain ecosystem is divided into multiple packages, allowing you to selectively install only the specific features or functionality you need.

To install the main langchain package, run on Python 3.11:

```
pip install langchain==0.3.20
```

Although this package serves as a good starting point for using LangChain, its real value lies in integrating with various model providers and datastores. The necessary dependencies for these integrations are not included by default and must be installed separately. The steps to do so are provided below.

The LangChain ecosystem consists of different packages designed for modular functionality, most of which rely on "langchain-core." This package includes base classes and abstractions, providing a foundation for the rest of the ecosystem. When installing any package, you don't need to explicitly install its dependencies like "langchain-core." However, if you need features from a specific version, you may do so, ensuring compatibility with other integrations.

Packages Overview

- **LangChain Core:** Contains essential abstractions and LangChain Expression Language (LCEL). Automatically installed with "langchain" or separately with

  ```
  pip install langchain-core==0.3.41
  ```

- **Integration Packages:** Packages like "langchain-openai" or "langchain-anthropic" offer support for specific integrations. The complete list of these integrations can be found under the "Partner libs" section in the API reference of the LangChain documentation. To install any of them, use

  ```
  pip install langchain-openai==0.3.7
  ```

  Integrations that haven't been split into their own packages are part of "langchain-community," installed via

  ```
  pip install langchain-community==0.3.19
  ```

- **Experimental Package:** "langchain-experimental" hosts research and experimental code. You can install it with

  ```
  pip install langchain-experimental==0.3.4
  ```

- **LangGraph:** A library designed for building stateful, multiactor applications with LLMs, which integrates seamlessly with LangChain but can be used independently:

  ```
  pip install langgraph==0.3.5
  ```

- **LangServe:** A tool to deploy LangChain runnables and chains as REST APIs. It is included with the LangChain CLI. If you need both client and

server functionalities, install using

```
pip install "langserve[all]"
```

For just the client or server, use `"langserve[client]"` or `"langserve[server]"`.

- **LangChain CLI:** Useful for managing LangChain templates and LangServe projects:

```
pip install langchain-cli==0.0.35
```

- **LangSmith SDK:** Installed automatically with "langchain" but does not depend on "langchain-core." It can be used separately if you're not using LangChain:

```
pip install langsmith==0.3.12
```

Installing from Source

To install any package from the source, clone the LangChain repository, navigate to the specific package's directory (e.g., "PATH/TO/REPO/langchain/libs/{package}"), and run

```
pip install -e .
```

This allows for flexible and targeted functionality, letting you selectively integrate or develop with specific packages in the ecosystem.

How to Prompt?

When working with large language models (LLMs), **prompt engineering** becomes an essential skill. A well-crafted prompt can significantly enhance the quality of a model's output, even when using less powerful or open source models. By understanding how to shape inputs effectively, you can guide LLMs to produce accurate, context-appropriate responses. Throughout this module, we'll explore the art and science of prompt creation, enabling you to fully harness the power of your models and achieve the best results possible.

One of the primary focuses will be on writing tailored prompts to achieve specific tasks, such as generating responses in a certain format or adhering to stylistic guidelines. We'll also examine how few-shot prompts can allow a model to quickly learn new tasks and generalize to unseen scenarios. This technique is especially useful when you need customization with minimal data, as it provides an efficient way to adapt model behavior on the fly.

Prompt Engineering

Prompt engineering is an emerging field focused on developing and refining prompts for effective use of large language models (LLMs) across a variety of applications. The goal is to enhance how LLMs process, understand, and generate text, making prompt engineering essential for numerous NLP tasks. Crafting high-quality prompts can reveal both the po-

tential and the boundaries of what LLMs can achieve, and a well-designed prompt can significantly improve the accuracy and relevance of the model's responses.

Throughout this lesson, you'll gain hands-on experience with practical examples, helping you understand the nuances of prompt quality. We'll explore how different prompts can lead to significantly different results, highlighting what makes a prompt "good" or "bad." By the end, you'll be equipped with techniques to create powerful prompts that enhance model performance, enabling it to provide contextually relevant, accurate, and insightful responses to any given task.

Role Prompting

Role prompting is a technique that asks an LLM to take on a specific role or persona, helping guide its response in line with a certain tone, style, or perspective. For example, you might prompt the model to act as a *"copywriter," "teacher," or "data analyst."* This provides the LLM with a frame of reference, shaping how it interprets and answers the prompt.

To work effectively with role prompting, follow these steps:

- **Define the Role Clearly:** Clearly specify the role in your prompt to set the context for the model. For example, you might write: "As a copywriter, craft catchy taglines for AWS services that grab attention." The model will interpret the role and respond accordingly, adopting the language and style of a copywriter.
- **Generate Output from the LLM:** Once the role is defined, use your prompt to produce an output. The model will use the role as guidance to tailor its response appropriately, focusing on the style, language, or structure that aligns with the defined role.
- **Iterative Refinement:** Analyze the output to see if it meets the desired criteria. If the results are not as expected, refine the prompt by being more specific about the role or the style of the response. This iterative process is crucial for achieving high-quality outputs. For example, if the response as a "copywriter" lacks creativity, you might adjust the prompt to include specific instructions like "use a playful tone and focus on benefits."

By guiding the model's behavior through role prompting, you can influence how it understands the task and the perspective it adopts, making it a versatile technique for a wide range of applications. This strategy not only improves the quality of the responses but also enables you to adapt the model's outputs to fit the context of different tasks more effectively.

NoteFor the following example, please get your OpenAI API key here: https://platform.openai.com/api-keys.

Example:

```
from langchain_core.prompts.prompt import PromptTemplate
from langchain_openai import ChatOpenAI
# Initialize the LLM with OpenAI's model
llm = ChatOpenAI(api_key=os.getenv("OPENAI_API_KEY"), model_name="gpt-4", temperature=0.5)
template = """
As a futuristic poet, I want to write a poem that captures the essence of {emotion}.
Can you suggest a title for a poem about {emotion} set in the year {year}?
"""
prompt = PromptTemplate(
```

```
    input_variables=["emotion", "year"],
    template=template,
)
# Input data for the prompt
input_data = {"emotion": "solitude", "year": "2500"}
chain = prompt | llm
response = chain.invoke(input_data)
print("Emotion: solitude")
print("Year: 2500")
print("AI-generated poem title:", response)
```

Output:

```
Emotion: solitude
Year: 2500
AI-generated poem title: content='"Echoes in the Void: Solitude in the 26th Century"' additional
```

The prompt in this code is effective for several reasons:

1. **Clear and Contextual Role Setting**
   By stating, "As a futuristic poet," the prompt establishes a role and context. This framing helps guide the model to think creatively like a poet, shaping its response to reflect a poetic tone and futuristic theme. Such context allows the LLM to adopt the right style, making the output more imaginative and relevant.
2. **Specificity of Emotion and Time Frame**
   The prompt specifically asks for a poem title that captures the emotion of "{emotion}" set in the year "{year}." This precision helps the model generate contextually rich and emotionally relevant titles, directly related to the emotion and future scenario. The use of variables makes it adaptable for different contexts, creating versatility.
3. **Open-Ended Creativity**
   The prompt is open-ended, allowing the LLM to generate diverse, creative titles without being overly restrictive. By not setting limitations on how the title should sound, the model can explore artistic and evocative language, enhancing the quality of the output.
4. **Task-Focused Guidance**
   The primary task is to create a poem title that evokes a specific emotion in a futuristic context. This direct focus helps the LLM avoid unrelated content, concentrating only on creating a unique title that matches the theme and style outlined in the prompt.
5. **Encouragement of Thematic Coherence**
   By guiding the LLM to align its output with an emotional and futuristic time frame, the prompt ensures the response will have both thematic and temporal coherence. This makes the resulting poem title not just relevant but also compelling and imaginative, showcasing how prompts can evoke specific styles and tones effectively.

Few-Shot Prompting

Few-shot prompting is a technique used in the context of large-scale language models to guide the model's output by providing a small number of task-specific examples within the input prompt. Unlike traditional machine learning approaches, which require extensive datasets and iterative training, few-shot prompting leverages a model's pre-existing knowledge to perform tasks with minimal supervision.

In few-shot prompting, the model is presented with a limited number of input/output pairs—usually between one and five—that illustrate the desired task. These examples serve as a form of implicit training within the prompt itself. The model uses these pairs to infer the relationship between inputs and outputs, allowing it to generalize and respond appropriately to new, unseen queries that follow the same pattern.

This technique builds on the premise that large language models, trained on vast amounts of diverse text data, can generalize across different domains. By presenting a few examples, the model can adjust its behavior dynamically without the need for explicit retraining or fine-tuning. Few-shot prompting thus demonstrates the flexibility and contextual reasoning ability of such models, allowing them to perform a wide range of tasks from a minimal set of instructions.

The effectiveness of few-shot prompting depends largely on the model's capacity to understand and generalize from the examples provided. It is a powerful approach for tasks where extensive labeled data is not readily available, offering an efficient method for leveraging pretrained models in a variety of applications.

Key Benefits

- **No Additional Training**: You don't need to fine-tune the model; it can perform tasks based on the few examples given.
- **Adaptability**: It can handle multiple tasks by simply providing examples for different tasks.
- **Efficiency**: Fewer examples are needed compared to traditional training methods, making it a practical approach for many applications.

Few-shot prompting is especially effective with very large pretrained models like GPT-3, which have enough capacity to learn from minimal examples.

Example:

```
from langchain_core.prompts.few_shot import FewShotPromptTemplate
from langchain_core.prompts.prompt import PromptTemplate
from langchain_openai import ChatOpenAI
# Initialize the language model with specific settings
language_model = ChatOpenAI(
    api_key="sk-proj-056py5goMfqp8_g2gOgfhefr1HLriyWyP6erQJ4dQyi3D2HWBxJgCWrjWMbvMTJdvxHlzaWm11
    model_name="gpt-4o-mini",
    temperature=0
)
# Sample color-to-emotion associations
color_emotion_pairs = [
    {"color": "red", "emotion": "energy"},
    {"color": "blue", "emotion": "peace"},
    {"color": "green", "emotion": "growth"},
]
# Template for formatting examples in a structured way
example_structure = """
Color: {color}
Associated Emotion: {emotion}\n
"""
# Create the example prompt template
color_prompt_template = PromptTemplate(
    input_variables=["color", "emotion"],
    template=example_structure,
```

```
)
# Construct a few-shot prompt template using the color-emotion pairs
few_shot_color_prompt = FewShotPromptTemplate(
    examples=color_emotion_pairs,
    example_prompt=color_prompt_template,
    prefix="Here are a few examples demonstrating the emotions linked with colors:\n\n",
    suffix="\n\nNow, considering the new color, predict the associated emotion:\n\nColor: {input
    input_variables=["input"],
    example_separator="\n",
)
# Generate the final prompt for a new color input
final_prompt_text = few_shot_color_prompt.format(input="purple")
# Use the generated prompt and run it through the language model
final_prompt = PromptTemplate(template=final_prompt_text, input_variables=[])
prompt_chain = final_prompt | language_model
# Get the AI-generated response for the input color
model_output = prompt_chain.invoke({})
# Print the input color and its corresponding predicted emotion
print("Color: purple")
print("Predicted Emotion:", model_output.content)
```

Output:

```
Color: purple
Predicted Emotion: Color: purple
Associated Emotion: creativity
```

Alternating Human/AI Messages

This strategy involves using few-shot prompting with alternating human and AI responses. It's particularly useful for chat-based applications, as it helps the language model grasp the flow of conversation and generate contextually relevant replies.

Though this method excels in handling conversational dynamics and is simple to implement for chat applications, it is less adaptable for other types of use cases and works best with chat-specific models. However, alternating human and AI messages can be applied creatively, such as building a prompt to translate English into pirate language in a chat format.

Chain Prompting

Chain prompting is a technique where multiple prompts are linked together in a sequence, with the output of one prompt being used as the input for the next. This method allows for progressively refining or expanding the context of the interaction, enabling the model to handle more complex tasks or multistep reasoning.

Key Characteristics

1. **Sequential Flow**: The process involves feeding the output from one step directly into the next, enabling the model to "remember" and build upon previous information.
2. **Dynamic Adjustments**: At each step, new information can be introduced based on the model's prior responses, allowing for iterative improvements in the result.

Steps for Chain Prompting

1. **Initial Prompt**: Start by providing an initial prompt to generate a base response.
2. **Extract Information**: Identify relevant details or key elements from the generated output.
3. **New Prompt Construction**: Create a subsequent prompt using the extracted information, adding new context or instructions to refine the output further.
4. **Repeat Process**: Continue chaining prompts as necessary, each building on the last, until the desired final output is obtained.

Using Chain Prompting in LangChain

To implement chain prompting in LangChain, you can leverage its `PromptTemplate` class. This class simplifies the construction of prompts by allowing for dynamic input values, making it ideal for situations where prompts need to evolve based on previous answers.

- **PromptTemplate** enables you to
  - Build prompts that adapt dynamically to changing inputs, ensuring flexibility in prompt chains
  - Simplify the process of passing outputs from one step to the next by easily substituting variables or new context into each prompt

Additional Benefits

- **Complex Workflows**: Chain prompting allows for handling more advanced tasks that require multiple steps, such as multiturn conversations, solving multipart problems, or conducting research in stages.
- **Error Handling**: If an intermediate step yields an incomplete or ambiguous response, chain prompting enables you to adjust the following prompts to clarify or correct the issue.
- **Interactive Exploration**: This approach allows for a more exploratory dialogue, where each prompt can refine the context, helping to uncover deeper insights.

In LangChain, combining chain prompting with other techniques like **few-shot prompting** or **memory-based approaches** allows you to build complex, multistep systems that leverage the power of large language models effectively.

Example:

```
from langchain_core.prompts.prompt import PromptTemplate
from langchain_openai import ChatOpenAI
# Initialize the language model
llm = ChatOpenAI(api_key="sk-proj-056py5goMfqp8_g2gOgfhefr1HLriyWyP6erQJ4dQyi3D2HWBxJgCWrjWMbvM
                 model_name="gpt-4o-mini",
                 temperature=0)
# Prompt 1: Ask for the scientist who developed the theory of general relativity
question_template = """Who is the scientist that formulated the theory of general relativity?
Answer: """
prompt_for_scientist = PromptTemplate(template=question_template, input_variables=[])
# Prompt 2: Ask for a brief explanation of the scientist's theory of general relativity
fact_template = """Give a brief explanation of {scientist}'s theory of general relativity.
Answer: """
```

```
prompt_for_fact = PromptTemplate(input_variables=["scientist"], template=fact_template)
# Create a runnable chain for the first prompt to retrieve the scientist's name
chain_for_question = prompt_for_scientist | llm
# Get the response for the first question
response_to_question = chain_for_question.invoke({})
# Extract the scientist's name from the response
scientist_name = response_to_question.content.strip()
# Create a runnable chain for the second prompt using the extracted scientist's name
chain_for_fact = prompt_for_fact | llm
# Input data for the second prompt
fact_input = {"scientist": scientist_name}
# Get the response for the second question about the theory
response_to_fact = chain_for_fact.invoke(fact_input)
# Output the scientist's name and the explanation of their theory
print("Scientist:", scientist_name)
print("Theory Description:", response_to_fact)
```

Output:

```
Scientist: The scientist who formulated the theory of general relativity is Albert Einstein.
Theory Description: content="Albert Einstein's theory of general relativity, formulated in 1915
```

Chain-of-Thought Prompting

Chain-of-thought prompting (CoT) is a technique designed to encourage large language models (LLMs) to explain their reasoning process, leading to more accurate outcomes. By presenting few-shot examples that showcase step-by-step reasoning, CoT helps guide the model to articulate its thought process when responding to prompts. This method has proven effective for tasks such as arithmetic, common sense reasoning, and symbolic logic.

In the context of LangChain, CoT offers several advantages. First, it helps deconstruct complex problems by guiding the model to break them into smaller, more manageable steps, which makes the problem easier to solve. This is especially useful for tasks involving calculations, logic, or multistep reasoning. Second, CoT can help the model generate more coherent and contextually relevant outputs by leading it through related prompts. This results in more accurate and meaningful responses, particularly for tasks that require deep comprehension of a problem or domain.

However, there are some limitations to CoT. One significant drawback is that it generally improves performance only when applied to models with approximately 100 billion parameters or more. Smaller models often generate illogical reasoning chains, which can result in lower accuracy compared to standard prompting. Additionally, CoT's effectiveness varies across different types of tasks. While it excels in tasks involving arithmetic, common sense, and symbolic reasoning, it may offer fewer benefits or even hinder performance in other task categories.

Advanced Tips for Effective Prompt Engineering

1. **Be Specific with Your Prompt**: Provide clear and detailed instructions in your prompt. The more context, background, and specifics you give, the better the LLM can interpret and generate a relevant response. Vague prompts lead to generalized or incomplete answers.

2. **Encourage Conciseness**: If the response needs to be short and to the point, be explicit about it. You can request responses to be limited to a specific number of words or sentences, which forces the model to focus on delivering the essential information.
3. **Ask for Reasoning or Explanations**: When dealing with complex tasks, encourage the model to explain its reasoning or show the steps it took to arrive at its answer. This improves the quality of results, particularly for problem-solving, logic, and reasoning tasks, ensuring transparency in the process.
4. **Iterate and Refine Prompts**: Prompt engineering is rarely a one-time activity. Iteration is key—test and tweak your prompts to see how different phrasing or added details change the model's response. Refine until the output aligns with your expectations.
5. **Use Examples to Guide Responses**: One of the most powerful ways to guide LLMs is by using few-shot learning. By showing the model a few examples of what you're looking for, you significantly increase the chance of receiving an answer that mirrors your expectations in tone, format, or reasoning.
6. **Apply Constraints**: If you're looking for specific formats or a particular structure (e.g., bulleted lists, headings, step-by-step processes), be clear about these constraints in your prompt. This helps the model organize its output according to your needs.
7. **Task-Specific Prompting**: Tailor your prompts to the specific task at hand. For example, creative writing prompts should encourage open-ended responses, while technical prompts should focus on precision, structure, and accuracy. Each type of task may require a different approach to prompt engineering.
8. **Leverage Clarifying Questions**: If the initial response isn't what you expected, ask the model to elaborate or clarify specific points. This helps guide the conversation in a more meaningful direction and ensures the model understands and focuses on what's important.
9. **Balance Open-Endedness and Constraints**: For tasks where creativity is needed, such as brainstorming, use more open-ended prompts to allow the model to explore a variety of ideas. For tasks requiring accuracy, use tighter constraints to keep the model focused on relevant and correct answers.
10. **Adjust Prompt Length**: The length of your prompt can influence the quality of the response. For some tasks, a simple, concise prompt works best, while more complex tasks might require detailed, multipart instructions. Experiment with prompt length to see what works for different types of questions.
11. **Include Key Terms**: If your task requires specific technical language, jargon, or domain-specific terms, include those directly in the prompt. This helps guide the model toward more specialized and accurate outputs, especially in fields like science, technology, or law.
12. **Specify the Role of the LLM**: Sometimes, framing the model's role in the prompt can improve the result. For instance, start your prompt with phrases like "As a teacher," or "You are an expert in…" to influence the model's tone and style of response, aligning it with the required task.
13. **Set an Output Persona**: In certain tasks, you can request the model to assume a specific persona or tone. For example, ask the model to respond like a teacher, researcher, or customer service agent to tailor the responses to different contexts or audiences.

14. **Utilize Multiturn Dialogue**: For tasks that require deeper exploration, consider breaking the problem down into a series of smaller questions. This approach not only helps the model focus on individual components of a complex task but also provides you with an opportunity to guide the conversation progressively toward a complete answer.
15. **Test Edge Cases**: For robustness, test how your prompt performs with edge cases or atypical inputs. This helps ensure that the LLM performs well across a variety of scenarios and doesn't generate inaccurate or nonsensical results in unusual situations.
16. **Account for Model Limitations**: Remember that LLMs have limitations in their knowledge and reasoning capabilities. Not all prompts will yield perfect responses, and some answers might lack depth or accuracy in certain specialized domains. Recognize when an LLM has reached its limit, and avoid overrelying on it for highly specialized or sensitive tasks.
17. **Keep Bias in Check**: Be mindful of the potential for biases in LLM-generated outputs. Craft prompts that minimize the chances of generating biased, harmful, or inappropriate content. Avoid phrasing that could steer the model toward biased or harmful assumptions.
18. **Incorporate Multiple Prompt Variations**: Instead of relying on one version of a prompt, try asking the same question or requesting the same task using several different prompt phrasings. This technique helps in uncovering new insights or variations in response quality.

By applying these strategies, you can enhance your ability to interact effectively with large language models, improving the quality and relevance of their outputs. As AI tools continue to evolve, mastery of prompt engineering will remain a critical skill for developers, researchers, and professionals who rely on LLMs to optimize their workflows.

What Are Chains?

A LangChain chain is a structured sequence of operations in the LangChain framework, where various components like language models, tools, and external APIs are connected to perform complex tasks. The primary purpose of a chain is to manage and coordinate interactions between different modules, allowing for multistep reasoning and advanced workflows when working with large language models (LLMs).

Key characteristics of a LangChain chain:

- **Modular Design:** Chains are designed to be modular, meaning individual components can be easily added, removed, or replaced. This allows for flexibility in constructing workflows depending on the use case, from simple to highly sophisticated tasks. Each module or component typically has a clearly defined input/output structure.
- **Multistep Processing:** Chains facilitate multistep operations by passing the output of one component as the input to another. This enables more advanced reasoning, decision-making, or actions that require several stages of processing, such as combining language understanding with tool execution or validation.
- **Control Flow:** Chains can incorporate control flow mechanisms, such as conditional logic or loops, enabling the workflow to branch or iterate based on the intermediate results. This allows for dynamic be-

havior, adjusting the sequence of actions depending on the inputs or outputs at each step.

- **Handling Intermediate Outputs:** A chain can retain intermediate outputs, either for logging purposes, debugging, or as part of a larger workflow. This allows for transparency in the process, making it easier to inspect how each step contributes to the final result.
- **Interaction with External Systems:** Chains are not limited to just working with language models. They can interact with external systems, such as databases, APIs, search engines, or knowledge bases, to fetch relevant information or execute tasks that go beyond natural language processing. This is particularly useful for retrieving real-time data, performing calculations, or executing functions that require interaction with other platforms.
- **Memory Management:** Some chains integrate memory, allowing them to store and recall past interactions, decisions, or context. This feature is particularly valuable for applications like conversational agents, where maintaining context over multiple interactions is critical for coherent and contextually aware responses.
- **Scalability:** Chains can be constructed in a scalable manner, allowing developers to design workflows that handle both simple tasks (such as a single prompt) or more intricate, multistep processes involving numerous components and external services.
- **Reusability:** LangChain encourages reusability by enabling the creation of reusable chains that can be applied to different tasks without reconfiguring the entire workflow. Developers can design a chain once and use it for various applications or modify it for similar tasks with minimal changes.

LangChain chains are an essential mechanism for building sophisticated applications that go beyond simple LLM queries, orchestrating complex interactions in a seamless, structured, and highly configurable way.

Chain Components

A LangChain chain consists of several key components that work together to create multistep workflows.

First, **prompt templates** are used to guide LLM outputs by filling in placeholders with dynamic values, helping customize the responses. The core of the system, **language models (LLMs)**, generate responses based on the input prompts. Chains can also integrate with **external tools**, such as APIs or databases, to fetch data or perform additional tasks beyond text generation.

Memory is another crucial component, allowing the chain to store and recall information across interactions, ensuring continuity, especially in conversational contexts. **Input variables** provide dynamic data that personalize the chain's behavior, while **output parsers** process and format model outputs for further steps or final responses.

More complex tasks can be handled by **nested chains (subchains)**, which break down workflows into smaller, manageable steps. **Decision logic** introduces conditional branching, enabling the chain to adapt based on input or intermediate results. Chains can also include **retrieval components** to fetch relevant information from external sources, enhancing context and accuracy.

Control flow governs the sequence and timing of operations, ensuring tasks are performed in the right order. To ensure robustness, **error handling mechanisms** are built in, managing failures and triggering retries or alternative steps when needed. **API connectors** allow chains to interact with external services, expanding functionality, while **logs and debugging** tools track execution, helping with monitoring and troubleshooting.

These components enable LangChain chains to integrate LLMs with tools, logic, and external data sources, allowing for flexible and complex workflows tailored to various applications.

Chain Types

In LangChain, there are several types of chains that can be used to construct workflows depending on the complexity, purpose, and specific requirements of the task. Each chain type serves a different function and can be adapted or combined to create versatile applications. Here are the most common types of **LangChain chains**.

1. Simple Chain

A simple chain consists of a single operation or a straightforward sequence of operations. This type of chain takes an input, processes it through one or more steps, and generates a single output. It's often used for basic tasks, such as filling in a prompt template and calling an LLM to generate a response.

- **Usage**: Direct question-answering tasks, summarization, or text transformation
- **Components**: Usually involves a single prompt template, one LLM call, and an output

Example:

```
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain_openai import OpenAI
# Step 1: Define the language model (in this case, OpenAI's GPT)
llm = OpenAI(api_key="sk-proj-056py5goMfqp8_g2gOgfhefr1HLriyWyP6erQJ4dQyi3D2HWBxJgCWrjWMbvMTJdv
            temperature=0.7)  # Set the desired temperature for response variability
# Step 2: Define the prompt template
prompt_template = """
Summarize the following question briefly:
{user_question}
"""
# Step 3: Create the PromptTemplate object
prompt = PromptTemplate(
   input_variables=["user_question"],
   template=prompt_template,
)
# Step 4: Create the LLMChain using the language model and prompt template
 chain = prompt | llm
# Step 5: Input the user's question and run the chain
user_question = "Can you explain how photosynthesis works in simple terms?"
output = chain.invoke(user_question)
# Print the summarized question
print("Summarized Question:", output)
```

Output:

```
Summarized Question:
Explaining photosynthesis in simple terms.
```

2. Sequential Chain

A sequential chain involves multiple steps arranged in a strict linear sequence. Each step's output becomes the input for the next step. These chains are useful when tasks need to be completed in a particular order.

- **Usage**: When multistep reasoning or progressive tasks are needed (e.g., generating an outline, followed by writing content based on that outline)
- **Components**: Multiple operations, such as LLM calls, external API interactions, or data transformations that occur in sequence

Example:

```
from langchain.chains import LLMChain, SimpleSequentialChain
from langchain.prompts import PromptTemplate
from langchain_openai import OpenAI
# Step 1: Define the language model
llm = OpenAI(api_key="sk-proj-056py5goMfqp8_g2gOgfhefr1HLriyWyP6erQJ4dQyi3D2HWBxJgCWrjWMbvMTJdv
            temperature=0.7)
# Step 2: Create the first prompt template to summarize the question
summary_prompt_template = """
Summarize the following question briefly:
{user_question}
"""
# Step 3: Create the second prompt template to generate a short answer
answer_prompt_template = """
Provide a brief answer to the following question:
{summarized_question}
"""
# Step 4: Create PromptTemplate objects for both prompts
summary_prompt = PromptTemplate(
    input_variables=["user_question"],
    template=summary_prompt_template,
)
answer_prompt = PromptTemplate(
    input_variables=["summarized_question"],
    template=answer_prompt_template,
)
# Step 5: Create LLMChain objects for both steps
summary_chain = LLMChain(llm=llm, prompt=summary_prompt)
answer_chain = LLMChain(llm=llm, prompt=answer_prompt)
# Step 6: Create a SimpleSequentialChain that links both chains together
sequential_chain = SimpleSequentialChain(
    chains=[summary_chain, answer_chain]
)
# Step 7: Input the user's question and run the sequential chain
user_question = "Can you explain how photosynthesis works in simple terms?"
output = sequential_chain.run(user_question)
# Print the output of the sequential chain
print("Final Output:", output)
```

Output:

```
Photosynthesis is the process by which plants and some other organisms use sunlight to turn wat
```

3. Conversational Chain

This chain is used in conversational agents where maintaining context is critical. It leverages memory to store and recall previous interactions, enabling the model to respond in a way that reflects the ongoing conversation.

- **Usage**: Chatbots, virtual assistants, customer support applications, or any system requiring multiturn conversations
- **Components**: LLMs for generating responses, memory for storing context, and potentially external tools for more complex interactions

> **Note**In the latest version of LangChain, you don't need to add the `openai_api_key` parameter anymore, but you need to define it as an environmental variable.

Example:

```
import os
# Set your OpenAI API key
os.environ["OPENAI_API_KEY"] = "sk-proj-056py5goMfqp8_g2gOgfhefr1HLriyWyP6erQJ4dQyi3D2HWBxJgCWr
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.memory import ConversationBufferMemory
from langchain.chains import LLMChain
# Step 1: Define a prompt template for conversation, using a variable for user input
prompt = ChatPromptTemplate.from_messages(
    [("user", "{user_input}")]
)
# Step 2: Set up the ChatOpenAI model (gpt-3.5-turbo in this case) with temperature control
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)
# Step 3: Create memory to store conversation history
memory = ConversationBufferMemory()
# Step 4: Create the chain combining prompt, model, and output parser
chain = LLMChain(prompt=prompt, llm=llm, memory=memory, output_parser=StrOutputParser())
# Simulate a conversation by invoking the chain with memory
# First user input
response_1 = chain.invoke({"user_input": "Can you explain what photosynthesis is?"})
print("AI Response 1:", response_1)
# Second user input
response_2 = chain.invoke({"user_input": "What happens during the light-dependent reactions?"})
print("AI Response 2:", response_2)
# Third user input
response_3 = chain.invoke({"user_input": "Can you summarize both for me?"})
print("AI Response 3:", response_3)
print(memory)
```

Output:

```
AI Response 1: {'user_input': 'Can you explain what photosynthesis is?', 'history': '', 'text':
AI Response 2: {'user_input': 'What happens during the light-dependent reactions?', 'history':
AI Response 3: {'user_input': 'Can you summarize both for me?', 'history': 'Human: Can you expla
chat_memory=InMemoryChatMessageHistory(messages=[HumanMessage(content='Can you explain what pho
```

4. Multi-input Chain

This type of chain accepts multiple inputs, which are processed either in parallel or in sequence depending on the workflow. It allows for more complex scenarios where various types of data or inputs must be handled together.

- **Usage**: When a task requires different sources of information, such as combining data from a user input and an external API or multiple models

- **Components**: Several input sources (e.g., a prompt and a knowledge base), multiple models, and tools to combine and process the inputs
  Example:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain.chains import SimpleSequentialChain
# Step 1: Define the first prompt to accept a question and context
question_prompt = ChatPromptTemplate.from_messages(
    [("user", "Given the context: '{context}', answer the question: '{question}'")]
)
# Step 2: Define the ChatOpenAI model
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)
# Step 3: Create the output parser
output_parser = StrOutputParser()
# Step 4: Combine the prompt and model into a chain
# This is a simple chain that handles multiple inputs (question and context)
chain = question_prompt | llm | output_parser
# Step 5: Define the inputs for the multi-input chain
inputs = {
    "question": "How does photosynthesis work?",
    "context": "Photosynthesis is the process used by plants to convert light energy into chemi
}
# Step 6: Run the chain with both inputs
response = chain.invoke(inputs)
# Output the response
print("Response:", response)
```

Output:

```
Response: Photosynthesis works by plants using sunlight to convert carbon dioxide and water int
```

Why Multi-input?

1. **Multi-input Prompt**: The `ChatPromptTemplate` defines a template that accepts two inputs: `context` and `question`. This prompt will insert both into the message for the language model.
2. **Model**: The `ChatOpenAI` model (`gpt-3.5-turbo`) is used to process the input and generate a response.
3. **Output Parser**: The `StrOutputParser` is used to parse the model's response into a string format. We will discuss the output parsers a bit later in the book.
4. **Chain Construction**: The chain combines the prompt, model, and output parser, handling both the question and context together as inputs.
5. **Invoke**: The `.invoke()` method is used to pass the inputs (both the question and the context) to the chain for processing.

5. Multi-output Chain

A multi-output chain takes an input and produces multiple outputs. This type of chain is useful when you want to generate different types of results based on a single input, such as extracting multiple pieces of information or generating multiple response options.

- **Usage**: Use cases where the same input must be processed in different ways, such as generating summaries, key takeaways, and action items from a single document
- **Components**: One input, multiple steps or LLM calls, and a set of outputs

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain.chains import LLMChain
from langchain.chains import SequentialChain
# Step 1: Define the prompt for generating a summary
summary_prompt = ChatPromptTemplate.from_messages(
    [("user", "Please summarize the following text: {input_text}")]
)
# Step 2: Define the prompt for extracting key points
key_points_prompt = ChatPromptTemplate.from_messages(
    [("user", "Extract the key points from the following text: {input_text}")]
)
# Step 3: Set up the ChatOpenAI model (same model for both tasks)
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)
# Step 4: Create the output parser
output_parser = StrOutputParser()
# Step 5: Create LLMChain for summarization and key point extraction
summary_chain = LLMChain(prompt=summary_prompt, llm=llm, output_key="summary")  # Changed output
key_points_chain = LLMChain(prompt=key_points_prompt, llm=llm, output_key="key_points")  # Chang
# Step 6: Create a SequentialChain that runs both chains (true multi-output)
multi_output_chain = SequentialChain(
    chains=[summary_chain, key_points_chain],
    input_variables=["input_text"],  # single input passed to both chains
    output_variables=["summary", "key_points"]  # two outputs
)
# Step 7: Define the input text
input_text = """
Photosynthesis is a process used by plants to convert light energy into chemical energy. During
plants take in carbon dioxide (CO2) and water (H2O) from the air and soil. Within the plant cel
meaning it loses electrons, while the carbon dioxide is reduced, meaning it gains electrons. Th
the water into oxygen and the carbon dioxide into glucose. The plant then releases the oxygen ba
and stores energy in the form of glucose molecules.
"""
# Step 8: Run the multi-output chain using apply() for multiple outputs
outputs = multi_output_chain.apply([{"input_text": input_text}])[0]
# Step 9: Output the responses
print("Summary:", outputs['summary'])
print("Key Points:", outputs['key_points'])
```

Output:

```
Summary: Photosynthesis is a process where plants convert light energy into chemical energy by
Key Points: - Photosynthesis is a process used by plants to convert light energy into chemical
- Plants take in carbon dioxide and water from the air and soil.
- Water is oxidized and carbon dioxide is reduced during photosynthesis.
- The result is oxygen and glucose production.
- Oxygen is released back into the air, while glucose is stored as energy in the plant.
```

Why It's a Multi-output Chain

1. **Single Input**: The input (input_text) is passed once and processed through multiple chains.

2. **Multiple Outputs**: The input is processed in two different ways (summary and key points), and the outputs are stored in distinct keys (summary, key_points).
3. **Sequential Execution**: The SequentialChain ensures that both chains run in sequence, with the same input generating multiple outputs in a single invocation.

Handling Multiple Outputs with apply():

- Since SequentialChain supports multiple output variables, we use apply() instead of run() to handle cases where more than one output is generated. This is essential for returning a dictionary with multiple output keys.

6. Router Chain

The router chain acts as a decision-making hub that directs the input to different subchains based on predefined conditions or classifications. It's useful when you have various workflows that depend on the type of input.

- **Usage**: For tasks requiring conditional logic, such as routing customer queries to the right department (billing, technical support, etc.) or choosing the right model based on input complexity
- **Components**: A router module that decides which subchain to invoke, along with those subchains themselves

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
# Step 1: Define the prompts for summarization and key points extraction
# Summarization prompt
summary_prompt = ChatPromptTemplate.from_messages(
    [("user", "Please summarize the following text: {input_text}")]
)
# Key points extraction prompt
key_points_prompt = ChatPromptTemplate.from_messages(
    [("user", "Extract the key points from the following text: {input_text}")]
)
# Classifier prompt to determine if the input is asking for a "summary" or "key points extracti
classifier_prompt = ChatPromptTemplate.from_messages(
    [("user", "Classify this request as 'summarization' or 'key points extraction': {input_text
)
# Step 2: Define the language model
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)
# Step 3: Define chains using the pipe operator
# Chain for classifying input
classifier_chain = classifier_prompt | llm | StrOutputParser()
# Chain for summarization
summary_chain = summary_prompt | llm | StrOutputParser()
# Chain for key points extraction
key_points_chain = key_points_prompt | llm | StrOutputParser()
# Step 4: Define a function to handle the routing based on classification
def router_chain(input_text):
    # Classify the input (is it a request for summarization or key points?)
    classification = classifier_chain.invoke({"input_text": input_text})
    # Route to the appropriate chain based on the classification result
    if "summarization" in classification.lower():
        return summary_chain.invoke({"input_text": input_text})
    elif "key points extraction" in classification.lower():
        return key_points_chain.invoke({"input_text": input_text})
    else:
        # Fallback to the summary chain if the classification is unclear
```

```
        return summary_chain.invoke({"input_text": input_text})
# Step 5: Define input texts
input_text_1 = "Summarize this text: Photosynthesis is a process used by plants to convert ligh
input_text_2 = "Give me the key points of the following text: Photosynthesis is a process used b
# Step 6: Run the router chain on different inputs
output_1 = router_chain(input_text_1)
output_2 = router_chain(input_text_2)
# Step 7: Print the outputs
print("Output 1:", output_1)
print("Output 2:", output_2)
```

Output:

```
Output 1: Photosynthesis is the process that plants use to convert light energy into chemical en
Output 2: - Photosynthesis is a process used by plants
- Plants convert light energy into chemical energy through photosynthesis
```

Why Router Chain?

- **Prompt Definition:** Each prompt is defined using ChatPromptTemplate.from_messages(). This includes the summary_prompt, key_points_prompt, and classifier_prompt for routing.
- **Chained Operations:** The chains (classifier_chain, summary_chain, and key_points_chain) are created using the pipe (|) operator to chain together the prompt, model (ChatOpenAI), and output parser (StrOutputParser).
- **Router Function:** The router_chain function first invokes the classifier_chain to classify the input as either a “summarization” or “key points extraction” task.
- Based on the classification result, it dynamically routes the input to the appropriate chain (summary_chain or key_points_chain). If the classification is unclear, it defaults to the summarization chain.
- **Running the Chain:** The router_chain function is run on two different inputs, input_text_1 and input_text_2, and the outputs are printed.

7. Control Flow Chain

A control flow chain allows branching and conditional execution based on the results of intermediate steps. The workflow can change dynamically depending on the decisions made at each stage, enabling complex reasoning processes.

- **Usage**: Scenarios where certain actions are taken only if specific conditions are met, such as checking the confidence level of a model’s output or validating an API response
- **Components**: Logic that governs branching (e.g., if-else statements), conditional steps, and error handling mechanisms

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
# Step 1: Define prompts for different tasks
# Prompt to answer a definition-related question
definition_prompt = ChatPromptTemplate.from_messages(
    [("user", "Define the following concept: {concept}")]
)
# Prompt to perform a calculation
calculation_prompt = ChatPromptTemplate.from_messages(
    [("user", "Calculate the following: {calculation}")]
)
```

```
# Classifier prompt to determine if the input is asking for a "definition" or a "calculation"
classifier_prompt = ChatPromptTemplate.from_messages(
    [("user", "Classify this request as 'definition' or 'calculation': {input_text}")]
)
# Step 2: Set up the ChatOpenAI model
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)
# Step 3: Define chains using the pipe operator
# Chain for classifying input
classifier_chain = classifier_prompt | llm | StrOutputParser()
# Chain for definition tasks
definition_chain = definition_prompt | llm | StrOutputParser()
# Chain for calculation tasks
calculation_chain = calculation_prompt | llm | StrOutputParser()
# Step 4: Define a function to handle control flow based on classification
def control_flow_chain(input_text):
    # Classify the input (is it a request for a definition or a calculation?)
    classification = classifier_chain.invoke({"input_text": input_text})
    # Route to the appropriate chain based on the classification result
    if "definition" in classification.lower():
        concept = input_text.replace("Define", "").strip()
        return definition_chain.invoke({"concept": concept})
    elif "calculation" in classification.lower():
        calculation = input_text.replace("Calculate", "").strip()
        return calculation_chain.invoke({"calculation": calculation})
    else:
        # Default response if classification is unclear
        return "Sorry, I didn't understand your request."
# Step 5: Define input texts
input_text_1 = "Define photosynthesis"
input_text_2 = "Calculate 5 + 3"
# Step 6: Run the control flow chain on different inputs
output_1 = control_flow_chain(input_text_1)
output_2 = control_flow_chain(input_text_2)
# Step 7: Print the outputs
print("Output 1:", output_1)
print("Output 2:", output_2)
```

Output:

```
Output 1: Photosynthesis is the process by which green plants, algae, and some bacteria convert
Output 2: 5 + 3 = 8
```

Key Features of the Control Flow Chain

1. **Conditional Logic**: The input is processed using conditional logic to determine which chain (definition or calculation) should handle the request.
2. **Dynamic Routing**: Based on the classification result, the input is dynamically routed to the appropriate chain.
3. **Flexible Task Handling**: This control flow chain can easily be extended to handle more types of inputs, making it a versatile way to manage tasks based on user requests.

8. Retrieval-Aware Chain

This chain is integrated with a retrieval mechanism, such as a vector database or a search engine, to retrieve relevant information before making decisions or generating responses. It's typically used in situations where context or additional data is needed to complete the task.

- **Usage**: Question-answering systems that need to pull information from knowledge bases or document repositories to provide accurate answers
- **Components**: A retrieval component (e.g., vector search or document retrieval) combined with LLM calls to process the retrieved information

> **Note**For the next example, you need to run the command `pip install faiss-gpu` as we use faiss.

Example:

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
# Step 1: Set up the FAISS vector store with embeddings
# This example assumes the OpenAI API is configured and available
# Define some documents (texts) related to quantum computing
documents = [
    "Quantum computing is a type of computation that harnesses the collective properties of quar
    "Quantum computers use quantum bits, or qubits, which can represent and store more informati
    "The fundamental principle of quantum computing is superposition, which allows qubits to be
    "Entanglement is another key property of quantum computing, allowing qubits to be interconne
]
# Step 2: Embed the documents using OpenAI embeddings
embeddings = OpenAIEmbeddings()  # Ensure you have OpenAI API keys configured
# Step 3: Create a FAISS vector store from the documents and their embeddings
vector_store = FAISS.from_texts(documents, embeddings)
# Step 4: Define the prompt that will use the retrieved context
retrieval_prompt = ChatPromptTemplate.from_messages(
    [("user", "Given the following context, answer the question: {context}")]
)
# Step 5: Define the ChatOpenAI model
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)
# Step 6: Define the retrieval-aware chain using FAISS
def retrieval_aware_chain(input_query):
    # Step 6.1: Retrieve relevant documents based on the query
    retrieved_documents = vector_store.similarity_search(input_query)  # FAISS similarity search
    context = " ".join([doc.page_content for doc in retrieved_documents])  # Combine documents
    # Step 6.2: Run the LLM chain with the retrieved context
    response = (retrieval_prompt | llm | StrOutputParser()).invoke({"context": context})
    return response
# Step 7: Define an input query
input_query = "What is quantum entanglement?"
# Step 8: Run the retrieval-aware chain
output = retrieval_aware_chain(input_query)
# Step 9: Print the output
print("Output:", output)
```

Output:

```
Output: What is entanglement in quantum computing?
Entanglement is a key property of quantum computing that allows qubits to be interconnected no r
```

9. Agent Chain

An agent chain is designed to allow a language model to interact with multiple tools or APIs autonomously. The LLM acts as an agent, deciding which tool to use and when, allowing for highly dynamic workflows where the model selects the appropriate actions.

- **Usage**: Complex applications where the model must autonomously decide which action to take, such as querying an API, searching a database, or executing a code snippet
- **Components**: The agent (LLM) interacts with external tools, APIs, or modules and follows predefined logic or dynamically generated plans

Example:

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.tools import Tool, BaseTool
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
import math
# Step 1: Define the tools (calculator and retrieval tool)
# Define a calculator tool to perform basic math operations
class CalculatorTool(BaseTool):
    def _run(self, input_query: str) -> str:
        """Run the calculator tool to perform basic arithmetic."""
        try:
            # Extract the mathematical expression by removing "calculate" or "Calculate"
            expression = input_query.lower().replace("calculate", "").strip()
            return str(eval(expression))  # Use eval safely for basic calculations
        except Exception:
            return "Invalid calculation."
    def name(self):
        return "calculator"
    def description(self):
        return "A simple calculator tool for performing basic arithmetic operations."
# Create an instance of CalculatorTool
calculator_tool = CalculatorTool()
# Define the FAISS-based retrieval tool for information retrieval
documents = [
    "Quantum computing is a type of computation that harnesses the collective properties of quar
    "Quantum computers use quantum bits, or qubits, which can represent and store more informati
    "The fundamental principle of quantum computing is superposition, which allows qubits to be
    "Entanglement is another key property of quantum computing, allowing qubits to be interconne
]
embeddings = OpenAIEmbeddings()
vector_store = FAISS.from_texts(documents, embeddings)
class RetrievalTool(BaseTool):
    def _run(self, input_query: str) -> str:
        """Run the retrieval tool to search the vector store for relevant information."""
        retrieved_documents = vector_store.similarity_search(input_query)
        return " ".join([doc.page_content for doc in retrieved_documents])
    def name(self):
        return "retrieval"
    def description(self):
        return "A tool for retrieving information about quantum computing."
# Create an instance of RetrievalTool
retrieval_tool = RetrievalTool()
# Step 2: Define the agent prompt with explicit instructions
agent_prompt = ChatPromptTemplate.from_messages(
    [("user", "If the query asks to perform a calculation (e.g., 'calculate 5 + 7'), respond wi
              "If the query asks for information (e.g., 'What is quantum computing?'), respond u
              "Input: {input_query}")]
)
# Step 3: Define the ChatOpenAI model (the agent)
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.7)
# Step 4: Define the agent chain function
def agent_chain(input_query):
    # Ask the agent to classify the task (calculation or retrieval)
    agent_decision = (agent_prompt | llm | StrOutputParser()).invoke({"input_query": input_query
    # Based on the agent's decision, invoke the appropriate tool
```

```
    if "calculate" in agent_decision.lower():
        return calculator_tool._run(input_query)
    else:
        return retrieval_tool._run(input_query)
# Step 5: Define the input queries
input_query_1 = "Calculate 5 + 7"
input_query_2 = "Explain quantum superposition"
# Step 6: Run the agent chain on different inputs
output_1 = agent_chain(input_query_1)  # Expecting a calculation result
output_2 = agent_chain(input_query_2)  # Expecting information retrieval result
# Step 7: Print the outputs
print("Output 1:", output_1)
print("Output 2:", output_2)
```

Output:

```
Output 1: 12
Output 2: The fundamental principle of quantum computing is superposition, which allows qubits
```

Breakdown of Key Steps of This More Complicated Code

- **CalculatorTool Definition:** A class CalculatorTool is defined, inheriting from BaseTool.
- **The _run() method is implemented, which**
  - Strips the word “calculate” from the input query.
  - Evaluates the remaining mathematical expression (e.g., “5 + 7”) using eval().
  - Returns the result of the calculation.
  - If the evaluation fails (e.g., for invalid expressions), it returns “Invalid calculation.”
- **RetrievalTool Definition**
  - A class RetrievalTool is defined, inheriting from BaseTool.
  - The _run() method is implemented, which
    - Uses FAISS to perform a similarity search based on the input query (e.g., “Explain quantum superposition”).
    - Retrieves relevant documents from the vector store.
    - Concatenates the content of the retrieved documents into a single response.
- **Embedding and Vector Store Setup**
  - A list of documents related to quantum computing is created.
  - OpenAIEmbeddings are used to embed these documents into vectors.
  - The document embeddings are stored in a FAISS vector store, which allows for similarity-based document retrieval.
- **Agent Prompt Setup**
  - **The agent prompt is defined, providing explicit instructions to the language model:**
    - If the input asks for a calculation (e.g., “Calculate 5 + 7”), the model should respond with “calculate.”
    - If the input asks for information (e.g., “What is quantum superposition?”), the model should respond with “retrieve.”

- **Agent Chain Function**
  - **The function agent_chain(input_query) performs the following steps:**
    - Passes the input query to the agent prompt (via the language model).
    - The agent responds with either “calculate” or “retrieve,” based on the task.
  - **Depending on the agent's decision:**
    - If “calculate” is returned, it calls the CalculatorTool to perform the calculation.
    - If “retrieve” is returned, it calls the RetrievalTool to fetch relevant information from the FAISS vector store.

10. Parallel Chain

A parallel chain allows multiple processes to run simultaneously, with their results combined at the end. This can improve efficiency when independent tasks can be processed at the same time.

- **Usage**: Situations where different tasks or models can be executed in parallel, such as generating multiple drafts of a text or performing several independent API calls
- **Components**: Multiple parallel operations that feed into a final aggregation or decision step

Example:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel
from langchain_openai import ChatOpenAI
# Step 1: Set up the OpenAI model
model = ChatOpenAI()
# Step 2: Define the chains for independent tasks
# Chain to summarize a concept
summarize_chain = ChatPromptTemplate.from_template("Summarize the concept of {concept}") | model
# Chain to provide detailed information about the concept
information_chain = ChatPromptTemplate.from_template("Provide detailed information about {concep
# Step 3: Set up the parallel chain to run both tasks concurrently
parallel_chain = RunnableParallel(summary=summarize_chain, information=information_chain)
# Step 4: Define the input concept
input_concept = {"concept": "Quantum computing"}
# Step 5: Run the parallel chain with the input concept
outputs = parallel_chain.invoke(input_concept)
# Step 6: Print the outputs
print("Summary Output:", outputs["summary"])
print("Information Output:", outputs["information"])
```

Output:

```
Summary Output: content='Quantum computing is a type of computing that utilizes the principles o
Information Output: content='Quantum computing is a type of computing that uses quantum-mechanic
```

What Does the Code Do?

1. **Model Setup**
   1. ChatOpenAI() is instantiated to serve as the language model for both tasks (summarization and detailed information retrieval).
2. **Chain Definitions**

1. **summarize_chain**: A prompt asks the model to summarize the given concept (e.g., “Quantum computing”).
2. **information_chain**: A prompt asks the model to provide detailed information about the same concept.

3. **Parallel Execution with RunnableParallel**
    1. **RunnableParallel** is used to execute both chains concurrently.
    2. Two chains are passed as arguments (summary for the summarization chain and information for the detailed information chain), which will run in parallel.
4. **Input Concept**
    1. The input concept is a dictionary containing the key “concept” with the value “Quantum computing.”
    2. This input is passed to both chains.
5. **Running the Chains in Parallel**
    1. The invoke() method is called on parallel_chain to execute both chains concurrently.
    2. The outputs are returned as a dictionary with keys “summary” and “information.”
6. **Outputs**
    1. The outputs from both chains (summary and detailed information) are printed.

Key Features of RunnableParallel

- **Concurrent Execution:** Both chains are executed concurrently, reducing the overall time required for execution.
- **Flexible Input Handling:** The same input (“Quantum computing”) is passed to both chains, but you can modify it to handle different inputs for each chain if needed.
- **Combined Outputs:** The results from both chains are combined into a single output dictionary, where each key corresponds to the respective chain’s output.

11. Custom Chain

Custom chains are tailored to the specific needs of an application, combining various components in novel ways. Developers can create a custom sequence of operations that fit their unique use case, combining steps from different chain types into a bespoke workflow.

- **Usage**: When none of the prebuilt chain types meet the specific requirements of the task, and custom logic, steps, or external integrations are needed
- **Components**: A combination of modules, tools, logic, and LLMs to suit the custom requirements of the application

These **LangChain chain types** provide a flexible framework for building diverse and sophisticated workflows tailored to the specific needs of different applications. By combining or modifying these chain types, developers can orchestrate complex interactions and achieve nuanced, multi-step tasks when working with large language models.

Example:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import Runnable
from langchain_openai import ChatOpenAI
```

```
# Step 1: Set up the OpenAI model
model = ChatOpenAI()
# Step 2: Define the chain for summarizing the concept
summarize_chain = ChatPromptTemplate.from_template("Summarize the concept of {concept}") | model
# Step 3: Define the chain for generating a quiz question based on the summary
quiz_chain = ChatPromptTemplate.from_template("Create a quiz question based on the summary: {su
# Step 4: Create a custom chain that first summarizes, then generates a quiz
class CustomChain(Runnable):
    def invoke(self, input_data):
        # First, get the summary of the concept
        summary = summarize_chain.invoke({"concept": input_data["concept"]})
        # Then, use the summary to generate a quiz question
        quiz_question = quiz_chain.invoke({"summary": summary})
        # Return both the summary and the quiz question
        return {"summary": summary, "quiz_question": quiz_question}
# Step 5: Create an instance of the custom chain
custom_chain = CustomChain()
# Step 6: Define the input concept
input_concept = {"concept": "Quantum computing"}
# Step 7: Run the custom chain with the input concept
output = custom_chain.invoke(input_concept)
# Step 8: Print the outputs
print("Summary Output:", output["summary"])
print("Quiz Question Output:", output["quiz_question"])
```

Output:

```
Summary Output: content='Quantum computing is a type of computing that uses quantum-mechanical p
Quiz Question Output: content='How does quantum computing utilize superposition and entanglement
```

Key Features of a Custom Chain

- **Custom Processing Logic:** The CustomChain class defines a two-step process: first generating a summary and then creating a quiz question based on the summary.
- **Sequential Execution:** The chain runs each step in sequence, passing the result of one step (summary) into the next step (quiz question generation).
- **Combined Outputs:** The chain returns both outputs (summary and quiz question) in a single response.

What Does This Code Do?

- **Model Setup:** Initializes a ChatOpenAI model to handle both summarization and quiz generation tasks
- **Summarization Chain:** Defines a chain (summarize_chain) that generates a summary of a concept based on a given input (e.g., “Quantum computing”)
- **Quiz Generation Chain:** Defines a chain (quiz_chain) that creates a quiz question based on the summary of the concept
- **CustomChain Class**
  - **Step 1:** Generates a summary of the concept using the summarize_chain
  - **Step 2:** Uses the summary to generate a quiz question with the quiz_chain
  - **Combined Output:** Returns both the summary and the quiz question as output
- **Execution:** Runs the custom chain by passing the concept (“Quantum computing”), and the chain outputs both a summary and a quiz question

- **Outputs:** Prints the generated summary and the quiz question based on the input concept

Conclusion

In this chapter, we covered the basics of LangChain and its integration with Python for building advanced NLP applications. We explored key components such as chains, prompts, memory, and tools, which enable developers to create flexible and scalable workflows. LangChain simplifies the process of working with large language models, allowing for efficient management of context and multistep processing.

By mastering these fundamental concepts, you are now equipped to build a variety of language model-based applications, from simple chatbots to more complex data retrieval systems.

In the next chapter, we'll dive deeper into more advanced components and conceptions like **LangChain Memory**, which enables models to retain information across interactions. We'll also explore **agents and tools** in LangChain, which allow dynamic decision-making, and discuss **indexes and retrievers**, essential for handling large datasets efficiently. These advanced features will help you build even more powerful and context-aware NLP applications.