LangChain Runnables: Primitives and Expression Language (LCEL)

1. The Evolution of Langchain Components and the Need for Runnables

Langchain initially provided a collection of components for building LLM applications, such as PromptTemplate for designing prompts, LLMs for interacting with models, Parsers for processing outputs, and Retrievers. While these components facilitated the creation of AI applications, a significant problem was their **lack of standardization**. Each component had its own unique way of interaction (e.g., format() for prompts, predict() for LLMs, parse() for parsers).

This inconsistency made it difficult to connect components seamlessly and build flexible workflows. Langchain identified this issue and decided to **standardize all components** to follow a common set of rules. The key standardization was the introduction of a universal invoke() function to interact with any component.

The technique used to achieve this standardization was **Runnables**.

"basically an abstract class was created and all these components, which are themselves classes, all these classes were inheriting that abstract class, and since they were inheriting that abstract class, you had to implement invoke and other methods in all these classes, which automatically standardized all your components."

This standardization allows components to be easily connected, where the output of one component automatically serves as the input for the next.

2. Types of Runnables

Runnables in Langchain are broadly categorized into two types:

2.1 Task-Specific Runnables

These are the **core Langchain components that have been converted into Runnables**. They each have a specific purpose.

"These are core LangChain components that have been converted into Runnables so that they can be used in pipelines."

Examples:

- ChatOpenAI: A component for interacting with OpenAI's chat models, now also a Runnable.
- PromptTemplate: For designing prompts, also a Runnable.
- Retriever: For retrieving documents, also a Runnable.

Their conversion to Runnables enables their use in pipelines and seamless connection with other Runnables.

2.2 Runnable Primitives

These are **fundamental building blocks designed to connect and orchestrate other Runnables, especially Task-Specific Runnables, to create complex AI workflows**.

"These are fundamental building blocks for structuring execution logic in AI workflows. They help orchestrate execution by defining how different Runnables interact sequentially, or in parallel, or conditionally, etc."

Runnable Primitives do not have a specific "task" in themselves; rather, their purpose is to define the flow and interaction between other Runnables.

Key Runnable Primitives discussed:

- **RunnableSequence**:
- **Purpose:** To connect two or more Runnables in a sequential manner. The output of the first Runnable becomes the input for the second, and so on.
- **Functionality:** Allows for the creation of chains where components execute one after another. There's no restriction on the number of Runnables that can be connected.
- **Analogy:** Similar to the custom "Runnable Connector" built in the previous video, but now a built-in Langchain class.
- **Example Use Case:** A chain where a PromptTemplate feeds into an LLM model, which then feeds into a StringOutputParser to generate a joke and then an explanation of the joke.
- **RunnableParallel**:
- **Purpose:** To execute multiple Runnables in parallel, each receiving the same input.
- **Functionality:**Each parallel branch operates independently.
- All branches receive the same input.
- The output is a dictionary, where keys typically represent the parallel tasks, and values are their respective outputs.
- **Example Use Case:** Generating both a tweet and a LinkedIn post about the same topic simultaneously using different LLM models or different prompts with the same model.
- **RunnablePassthrough**:
- **Purpose:** A special Runnable Primitive that simply **passes its input directly as its output without any modification or processing**.
- **Functionality:**Receives an input and returns it as is.
- Can be an integer, string, dictionary, or any data type.
- **Example Use Case:** Useful in scenarios where you need to retain a specific output from an earlier step in a chain while other parallel branches perform additional processing. For instance, generating a joke and its explanation in parallel, and RunnablePassthrough ensures the original joke content is preserved in the final output alongside the explanation.
- **RunnableLambda**:
- **Purpose:** To **convert any standard Python function into a Runnable**.
- **Functionality:**Allows custom Python logic (e.g., data preprocessing, specific calculations) to be integrated seamlessly into a Langchain chain.
- Once a Python function is converted to a Runnable, it can be connected with other Runnables in a sequence or parallel flow.
- **Example Use Case:**Pre-processing customer reviews before sending them to an LLM for sentiment analysis (e.g., removing HTML tags, punctuation, converting to lowercase).
- Counting the number of words in a generated joke alongside printing the joke itself.

- **RunnableBranch**:
- **Purpose:** To create **conditional chains**, acting as Langchain's equivalent of an "if-else" statement.
- **Functionality:**Evaluates conditions and executes a specific Runnable (or chain of Runnables) based on which condition is true.
- Takes a series of tuples, each containing a condition (a lambda function that evaluates to a boolean) and the Runnable to execute if that condition is met.
- Includes a default "else" condition that triggers if no other condition is true.
- **Example Use Case:** Processing customer emails differently based on their content (e.g., complaint, refund request, general query). In a detailed example, generating a report, then checking its length; if it's over 500 words, summarize it using an LLM; otherwise, print it as is using RunnablePassthrough.

3. Langchain Expression Language (LCEL)

LCEL is a **declarative way to define chains** in Langchain, specifically designed to simplify the creation of RunnableSequence chains.

- **Motivation:** The creators observed that RunnableSequence is a very common and frequently used primitive for building sequential chains.
- **Syntax:** Instead of explicitly instantiating RunnableSequence and passing a list of Runnables (e.g., RunnableSequence([R1, R2, R3])), LCEL allows the use of the **pipe (|) operator** for sequential chaining.
- "Rather than defining a Runnable Sequence like this, you can define it like this: R1 | R2 | R3."
- **Benefits:** Offers a more concise and readable syntax for defining sequential workflows.
- **Current State:** As of the video, LCEL primarily caters to RunnableSequence.
- **Future Prospects:** There is a high probability that future versions of Langchain will extend LCEL to provide declarative syntax for other Runnable Primitives as well (e.g., an & operator for RunnableParallel or specific syntax for RunnableBranch). This would further simplify the construction of complex AI workflows.