
LangGraph

Type Annotations

Dictionary 📖

Normal Dictionary:

```
movie = {"name": "Avengers Endgame", "year": 2019}
```

- Allows for efficient data retrieval based on unique keys
- Flexible and easy to implement
- Leads to challenges in ensuring that the data is a particular structure, especially for larger projects
- Doesn't check if the data is the correct type or structure

Typed Dictionary ⌨️

```python
from typing import TypedDict

class Movie(TypedDict):
    name : str
    year : int

movie = Movie(name="Avengers Endgame", year=2019)
```

- Type Safety - we defined explicitly what the data structures are, reducing runtime errors
- Enhanced Readability - Makes debugging easier and makes code more understandable.

Union 🤝

```python
from typing import Union

def square(x: Union[int, float]) -> float:
    return x * x

x = 5       # ✅ this is fine because it is an integer
x = 1.234   # ✅ this is also fine because it is a float
x = "I am a string!"    # ❌ this will fail because it is a string
```

- Union lets you say that a value can be more than one type
- Flexible and easy to code
- Type Safety as it can provide hints to help catch incorrect usage

Optional 🤔

```python
from typing import Optional

def nice_messasge(name: Optional[str]) -> None:
    if name is None:
        print("Hey random person!")
    else:
        print(f"Hi there, {name}!")
```

- In this case “name” can be either String or None!
- It cannot be anything else

Any 🎲

```python
from typing import Any

def print_value(x: Any):
    print(x)

print_value("I pretend to be Batman in the shower sometimes")
```

- Anything and everything is allowed!

Lambda Function ⏳

```
square = lambda x: x * x
square(10)
```

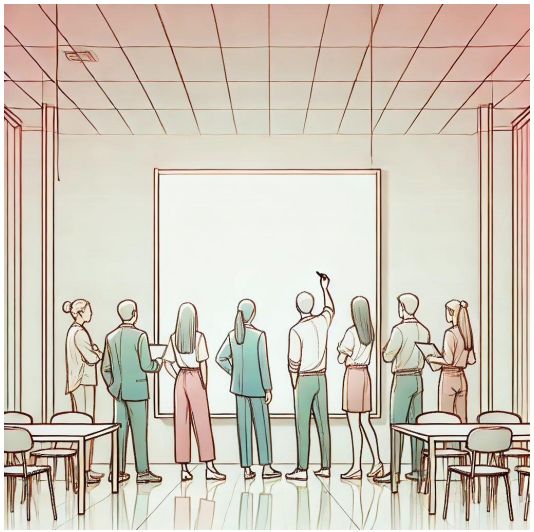
```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x * x, nums))
```

- Lambda is just a shortcut to writing small functions!

Elements

State 🧮

- ➔ The State is a shared data structure that holds the current information or context of the entire application.
- ➔ In simple terms, it is like the application's memory, keeping track of the variables and data that nodes can access and modify as they execute.

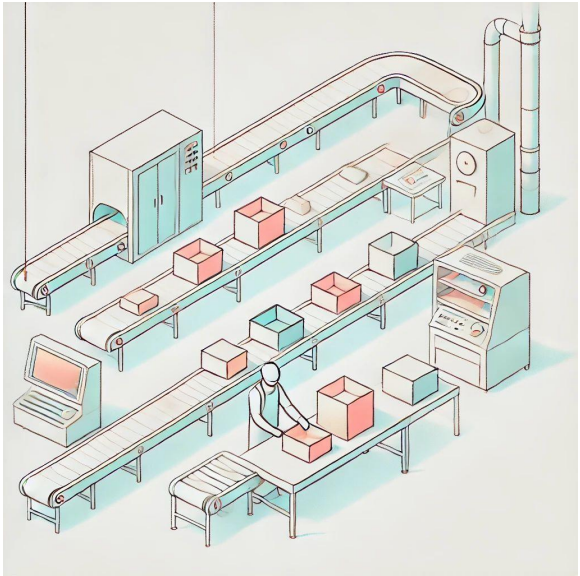


Analogy:

- **Whiteboard in a Meeting Room:** Participants (nodes) write and read information on the whiteboard (state) to stay updated and coordinate actions.

Nodes 📌

- ➔ **Nodes** are individual functions or operations that perform specific tasks within the graph.
- ➔ Each node receives input (often the current state), processes it, and produces an output or an updated state.

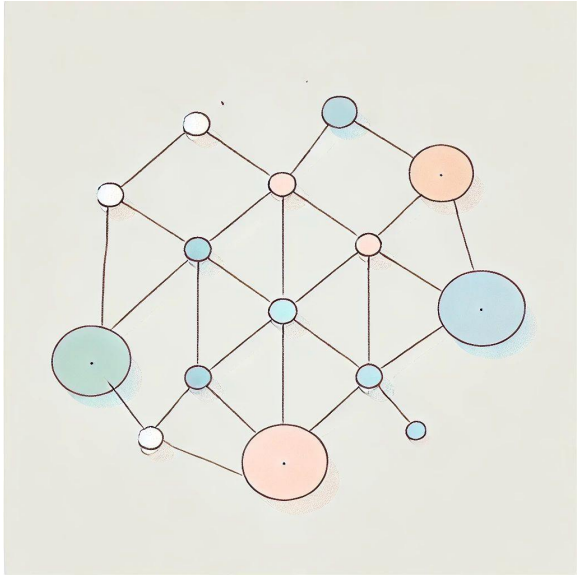


Analogy:

- **Assembly Line Stations:** Each station does one job—attach a part, paint it, inspect quality, and so on.

Graph 🍡

- ➔ A **Graph** in LangGraph is the overarching structure that maps out how different tasks (nodes) are connected and executed.
- ➔ It visually represents the workflow, showing the sequence and conditional paths between various operations.

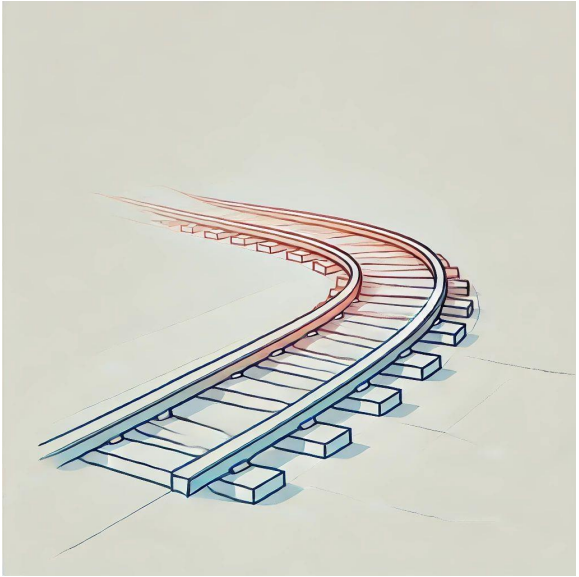


Analogy:

- **Road Map:** A road map displaying the different routes connecting cities, with intersections offering choices on which path to take next.

Edges 🔥

- ➔ **Edges** are the connections between nodes that determine the flow of execution.
- ➔ They tell us which node should be executed next after the current one completes its task.

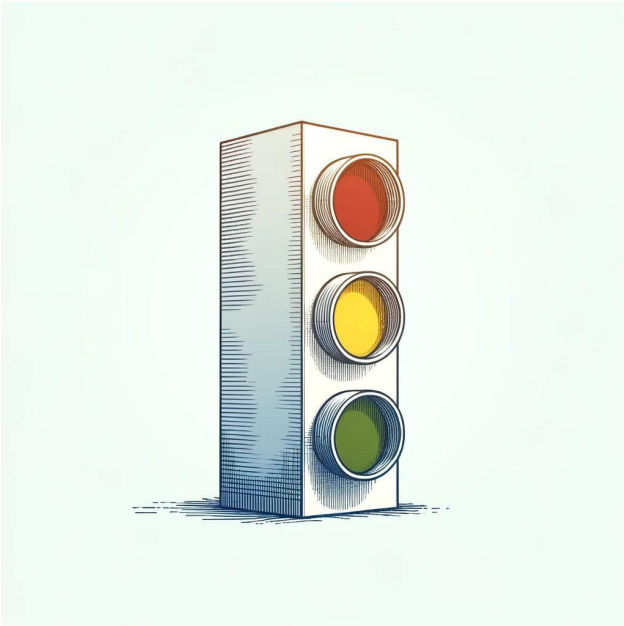


Analogy:

- **Train Tracks:** Each track (edge) connects the stations (nodes) together in a specific direction.

Conditional Edges 🎏

➔ **Conditional Edges** are specialized connections that decide the next node to execute based on specific conditions or logic applied to the current state.



Analogy:

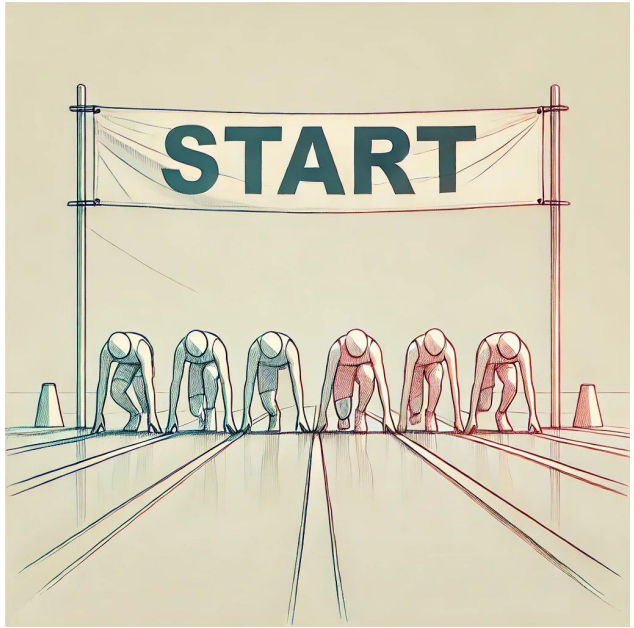
- **Traffic Lights:** Green means go one way, red means stop, yellow means slow down. The condition (light color) decides the next step.

START 🚀

- ➔ The **START** node is a virtual entry point in LangGraph, marking where the workflow begins.
- ➔ It doesn't perform any operations itself but serves as the designated starting position for the graph's execution.

Analogy:

- **Race Starting Line:** The place where a race officially begins.



END 🚩

- ➔ The **END** node signifies the conclusion of the workflow in LangGraph.
- ➔ Upon reaching this node, the graph's execution stops, indicating that all intended processes have been completed.

Analogy:

- **Finish Line in a Race:** The race is over when you cross it.

Tools 🛠️

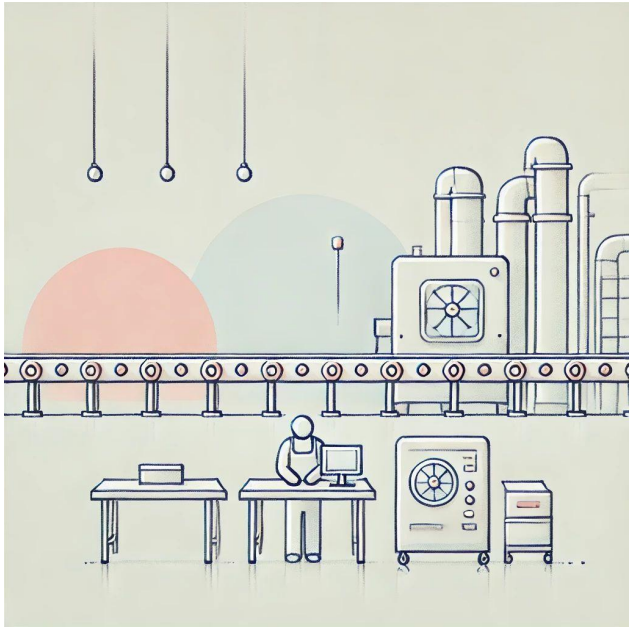
- ➔ **Tools** are specialized functions or utilities that nodes can utilize to perform specific tasks such as fetching data from an API.
- ➔ They enhance the capabilities of nodes by providing additional functionalities.
- ➔ Nodes are part of the graph structure, while tools are functionalities used within nodes

Analogy:

- **Tools in a Toolbox:** A hammer for nails, a screwdriver for screws, each tool has a distinct purpose.

ToolNode 🔎

- ➔ A **ToolNode** is just a special kind of node whose main job is to run a tool.
- ➔ It connects the tool's output back into the State, so other nodes can use that information.



Analogy:

- **Operator Using a Machine:** The operator (ToolNode) controls the machine (Tool), then takes the results back to the assembly line.

StateGraph 🏤

- ➔ A **StateGraph** is a class in LangGraph used to build and compile the graph structure.
- ➔ It manages the nodes, edges, and the overall state, ensuring that the workflow operates in a unified way and that data flows correctly between components.

Analogy:

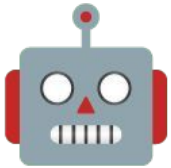
- **Blueprint of a Building:** Just as a blueprint outlines the design and connections within a building, a StateGraph defines the structure and flow of the workflow.

Messages 📬

Human Message
Represents input from a user.

System Message
Used to provide instructions or context to the model

Function Message
Represents the result of a function call



AI Message
Represents responses generated by AI models

Tool Message
Similar to Function Message, but specific to tool usage

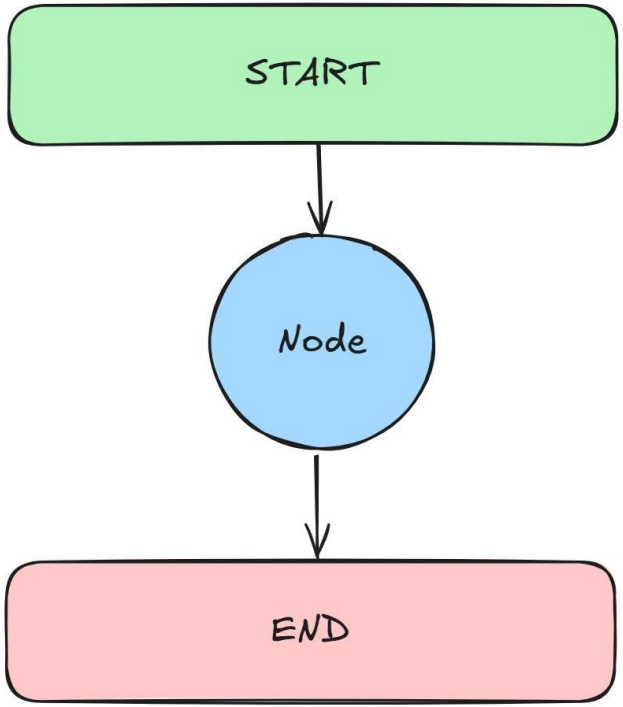
Graph I

not yet… but soon!

Hello World Graph 🌍

Objectives :✅

1. Understand and define the AgentState structure
2. Create simple node functions to process and update state
3. Set up a basic LangGraph structure
4. Compile and invoke a LangGraph graph
5. Understand how data flows through a single-node in LangGraph


START
Node
END

Graph II

Conditional Graph 

Objectives:

1. Implement conditional logic to route the flow of data to different nodes
2. Use **START** and **END** nodes to manage entry and exit points explicitly.
3. Design multiple nodes to perform different operations (addition, subtraction).
4. Create a router node to handle decision-making and control graph flow.

Main Goal: How to use “add_conditional_edges()”

Graph III

Looping Graph 

Objectives:

1. Implement looping logic to route the flow of data back to the nodes
2. Create a single conditional edge to handle decision-making and control graph flow.

Main Goal: Coding up Looping Logic

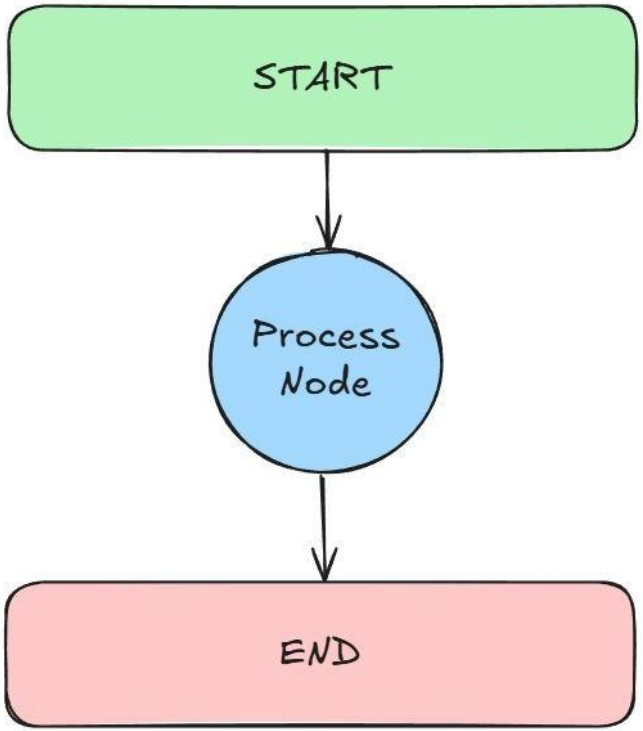
Agent I

Simple Bot 🤖

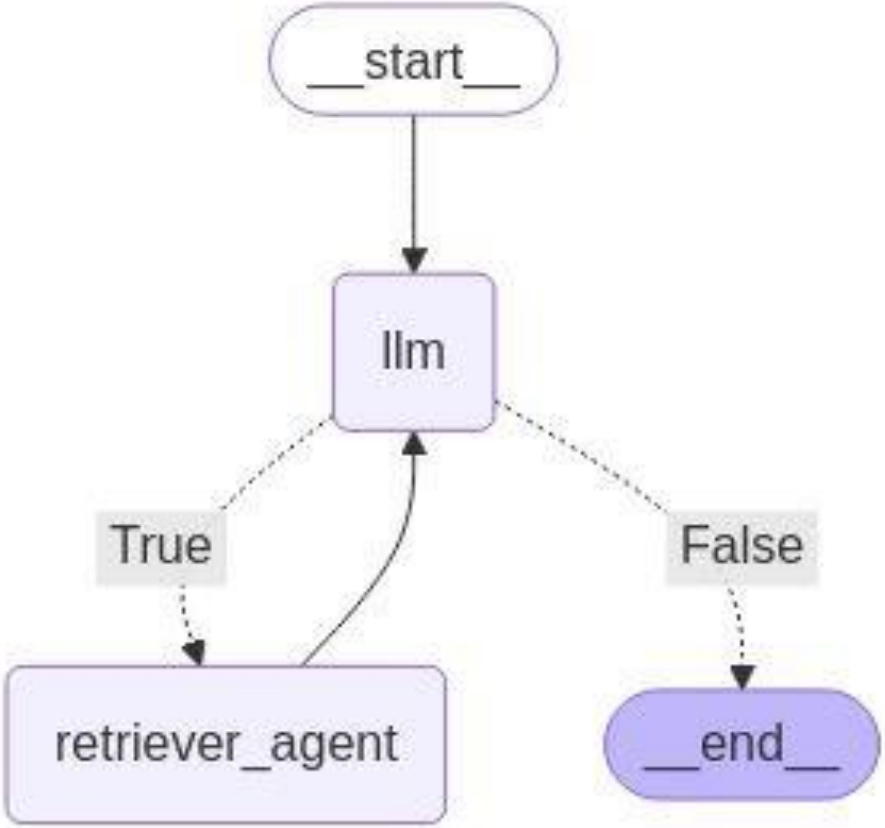
Objectives:

1. Define state structure with a list of HumanMessage objects.
2. Initialize a GPT-4o model using LangChain's ChatOpenAI
3. Sending and handling different types of messages
4. Building and compiling the graph of the Agent

Main Goal: How to integrate LLMs in our Graphs


START
Process Node
END

RAG


__start__
llm
True
False
retriever_agent
__end__