1. Simple Graph

Goal: Build a basic graph with conditional flow.

- Use `StateGraph`, `START`, `END` to control flow.
- Define nodes with functions to mutate or append state.
- Use `add_node`, `add_edge`, `add_conditional_edges`.

Key Code:

```
from typing_extensions import TypedDict
class State(TypedDict):
    graph_state: str

def node_1(state):
    return {"graph_state": state['graph_state'] +" I am"}

def node_2(state):
    return {"graph_state": state['graph_state'] +" happy!"}

def node_3(state):
    return {"graph_state": state['graph_state'] +" sad!"}

from langgraph.graph import StateGraph, START, END
builder = StateGraph(State)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

# Conditional routing
from typing import Literal
import random
def decide_mood(state) -> Literal["node_2", "node_3"]:
    return "node_2" if random.random() < 0.5 else "node_3"

builder.add_edge(START, "node_1")
builder.add_conditional_edges("node_1", decide_mood)
builder.add_edge("node_2", END)
builder.add_edge("node_3", END)
graph = builder.compile()
graph.invoke({"graph_state" : "Hi, this is Lance."})
```

---

2. Chain with LLM

Goal: Use LLM with tools in a LangGraph.

- Define `MessagesState` to manage chat context.
- Bind tools to LLM using `.bind_tools()`.
- Build a single-node graph that invokes LLM.

Key Code:

```
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o")
```

```python
def multiply(a: int, b: int) -> int:
    return a * b

llm_with_tools = llm.bind_tools([multiply])

from langgraph.graph import MessagesState, StateGraph, START, END
from langchain_core.messages import HumanMessage

def tool_calling_llm(state: MessagesState):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(MessagesState)
builder.add_node("tool_calling_llm", tool_calling_llm)
builder.add_edge(START, "tool_calling_llm")
builder.add_edge("tool_calling_llm", END)
graph = builder.compile()
```

---

3. Router

Goal: Use `tools_condition` to route LLM output.

- Add `ToolNode` and conditional edge to check if tool was called.

Key Code:

```python
from langgraph.prebuilt import ToolNode, tools_condition

def multiply(a: int, b: int) -> int:
    return a * b

llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools([multiply])

def tool_calling_llm(state: MessagesState):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(MessagesState)
builder.add_node("tool_calling_llm", tool_calling_llm)
builder.add_node("tools", ToolNode([multiply]))
builder.add_edge(START, "tool_calling_llm")
builder.add_conditional_edges("tool_calling_llm", tools_condition)
builder.add_edge("tools", END)
graph = builder.compile()
```

---

4. Agent (ReAct-style)

Goal: Loop between assistant and tools for multi-step reasoning.

- Implements ReAct loop using `tools_condition`.
- Adds bidirectional edge between `Assistant` and `ToolNode`.

Key Code:

```python
from langgraph.prebuilt import ToolNode, tools_condition

def add(a: int, b: int) -> int: return a + b
def multiply(a: int, b: int) -> int: return a * b
def divide(a: int, b: int) -> float: return a / b

tools = [add, multiply, divide]
llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools(tools, parallel_tool_calls=False)

from langchain_core.messages import SystemMessage
sys_msg = SystemMessage(content="You are a helpful assistant...")

def assistant(state: MessagesState):
   return {"messages": [llm_with_tools.invoke([sys_msg] + state["messages"])]}

builder = StateGraph(MessagesState)
builder.add_node("assistant", assistant)
builder.add_node("tools", ToolNode(tools))
builder.add_edge(START, "assistant")
builder.add_conditional_edges("assistant", tools_condition)
builder.add_edge("tools", "assistant")
react_graph = builder.compile()
```

---

5. Agent with Memory

Goal: Add persistent memory using `MemorySaver`.

- State is stored using a `thread_id`.
- Enables context retention across calls.

Key Code:

```python
from langgraph.checkpoint.memory import MemorySaver
memory = MemorySaver()
react_graph_memory = builder.compile(checkpointer=memory)

# Thread config
config = {"configurable": {"thread_id": "1"}}

messages = [HumanMessage(content="Add 3 and 4.")]
react_graph_memory.invoke({"messages": messages}, config)

messages = [HumanMessage(content="Multiply that by 2.")]
react_graph_memory.invoke({"messages": messages}, config)
```
