목차

**I.RAG,Function Calling,Runnable**

1. LLM,RAG,function calling

2. langchain,runnable,agent

**II. LangGraph**

3. Fundamentals of LangGraph

4. Stateful Graph

5. Conditional\_edge

6. Tool calling ( Function calling )

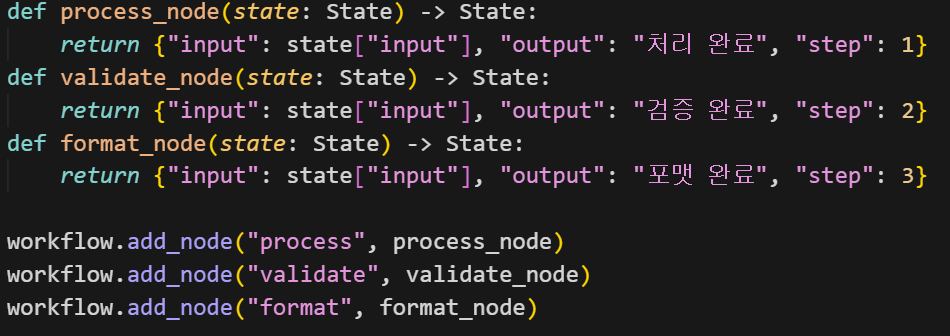
7. Prompt Engineering / Context Engineering

8. state,reducer / state,context,config

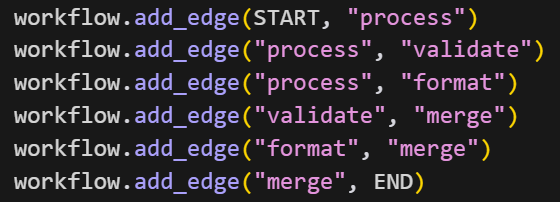
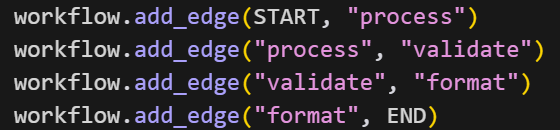
9. checkpoint,hil ( \*saver, interrupt, command)

**3. Fundamentals of LangGraph**

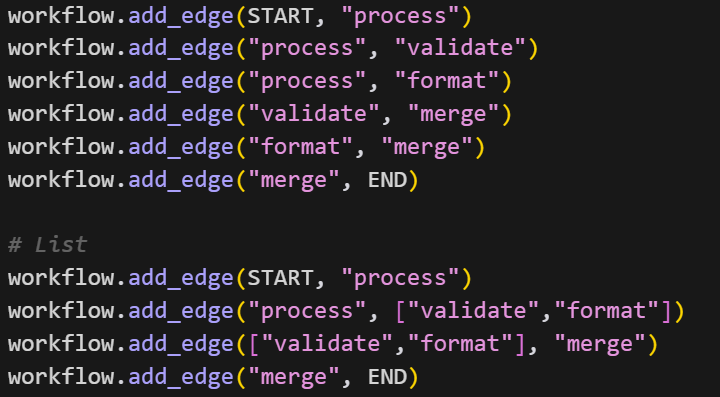
* graph : node,edge로 구성
* node : 함수를 node로 한번 더 선언 ( 선언순서와 실행순서는 무관 )
* edge : node를 이어서 순서를 만든다. ( 노드선언만으로는 동작하지 않음 )
* 그래프명 선언 -> 함수 선언 -> 함수를 노드로 선언 -> 엣지 연결 -> compile
* 그래프명 선언 후, 그래프명.add\_node(“노드명”, 함수 이름)



* 그래프명.add\_edge(START,”노드1”)
* 그래프명.add\_edge(“노드1”,”노드2”)
* 그래프명.add\_edge(“노드2”,END)
* 엣지는 직렬/병렬 가능 ( sequential,parallel : fan out/fan in )



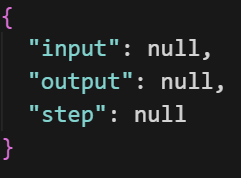
* fan out : (“node1”,[“node2,”node3”]) / (“node1”,”node2”) (“node1”,”node3”)
* fan in : ( [“node2”,”node3”],”node4”) / ( “node2,”node4”) (“node3”,”node4”)

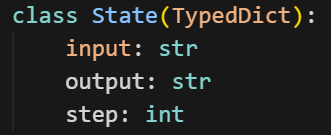


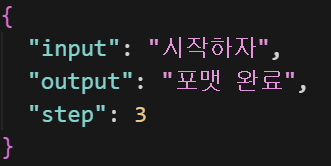
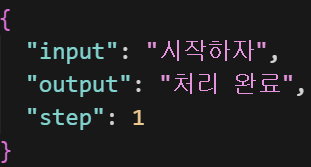
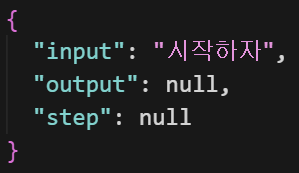
* compile() / app.invoke()



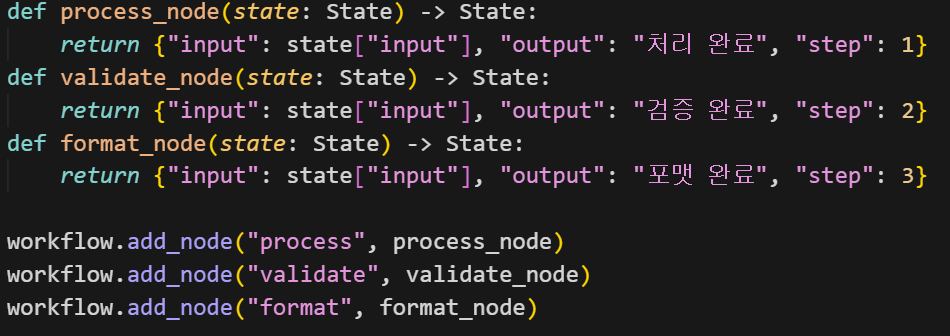
**4. Stateful Graph ( StateGraph )**

* 실행 중에 상태를 실제로 저장하고 업데이트하는 그래프
* node에서 전달되는 데이터, class로 선언, 상속
* state는 딕셔너리 형태 (Typeddict)
* 선언시 메모리에 state 딕셔너리 생성
* input은 모든 state / output 는 설정에 따라 다름

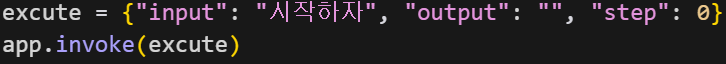




* 최근 트렌드는 stategraph 만 사용.
* 구조 : 그래프객채명 = StateGraph(상속받을 스테이트명)
* node에서 스테이트 상속, node1-node2-node3 … 메모리에 저장된 스테이트 상속

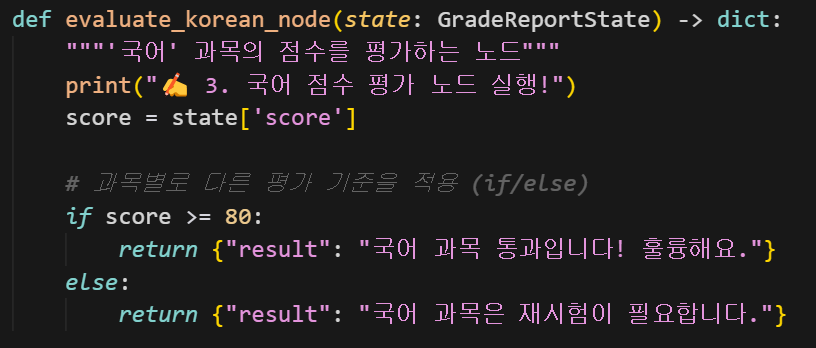


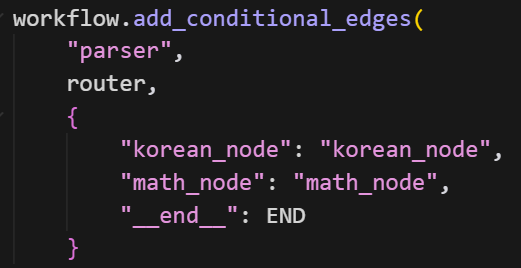
* invoke,stream 실행
* LangGraph 워크플로우를 시작할 때, State에 정의된 모든 항목을 포함하는 딕셔너리를 전달
* 아직 값이 없는 항목은 해당 타입에 맞는 초기값("", 0 등)으로 설정.

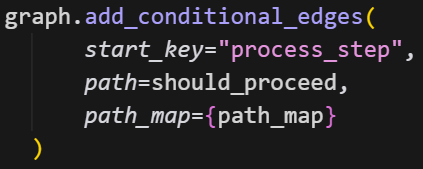


**5. Conditional\_edge**

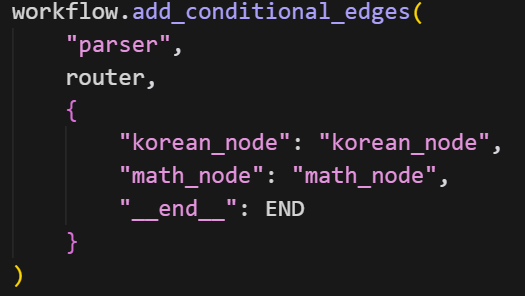
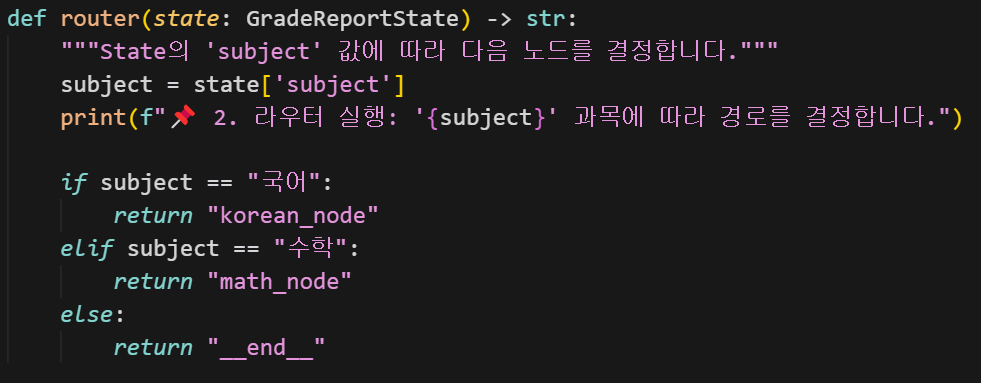
* 조건부 노드 : State는 노드 간에 공유되는 '데이터'이자, 노드 내부 로직의 '판단 근거'
* state data가 조건부 핵심이자 이정표



* 조건부 엣지 (Conditional\_edges)  
  : 그래프객체.add\_conditional\_edges(분기점\_노드명, 조건\_함수, 경로\_맵)



* 입력된 state를 바탕으로 노드경로를 정한다  
  : router - 국어 = korean\_node / 수학 = math\_node  
  : 노드1 - 라우터 ( 노드1의 입력된 정보에 따른 다음 노드 출력력 )

****

**6. Tool calling ( Function calling )**

* Prompt / Context Engineering
* LLM을 출력하여 TOOL을 결정한다.
* LLM을 시스템 프롬프트를 바탕으로 어떤 tool을 사용할지 tool\_name을 출력한다.
* 출력된 tool\_name은 state에 업데이트 된다.
* 다음 노드에서 state data를 바탕으로 tool을 호출한다.
* 조건부로 tool\_name 과 조건부를 출력하도록 한다. ( 예 날씨tool, 위치 서울 )
* 순서  
  : LLM의 판단 → State 업데이트 → 조건부 라우팅 → Tool 실행 → State 업데이트 → 답변
* 구현순서

**1단계: Agent 노드 (LLM이 Tool 사용 결정)**

* 사용자의 질문(Prompt)과 우리가 미리 정의해둔 사용 가능한 **도구(Tools) 목록**을 LLM에게 함께 전달합니다.
* LLM은 이 정보를 바탕으로 사용자의 질문에 답하기 위해 도구가 필요한지 판단합니다.
* **도구가 필요하다고 판단하면**, 어떤 도구를 어떤 인자(arguments)와 함께 호출해야 할지를 결정하여 **'Tool Call' 객체**를 반환합니다.
  + 예: tool\_name='get\_weather', arguments={'location': '서울'}
* 이 'Tool Call' 객체를 State에 업데이트하여 저장합니다.

**2단계: 조건부 라우터 (Conditional Edge)**

* Agent 노드 다음에는 조건부 엣지를 둡니다.
* 라우터 함수는 State를 확인하여 **'Tool Call' 객체가 있는지 없는지**를 판단합니다.
  + **Tool Call 있음**: 'Tool 실행' 경로로 분기합니다. ("tool\_executor")
  + **Tool Call 없음**: LLM이 바로 답변을 생성했다고 판단하고, '최종 답변' 경로 또는 END로 분기합니다.

**3단계: Tool Executor 노드 (실제 Tool 호출)**

* 라우터에 의해 'Tool 실행' 경로로 오게 되면, 이 노드가 실행됩니다.
* 이 노드는 State에 저장된 tool\_name과 arguments를 읽어옵니다.
* 읽어온 정보를 바탕으로 실제 파이썬 함수(도구)를 실행합니다. (예: get\_weather(location='서울') 함수 호출)
* 함수 실행 후 얻은 **결과**(예: "오늘 서울의 날씨는 맑음, 25도입니다.")를 다시 State에 업데이트하여 저장합니다.

**4단계: 다시 Agent 노드로 (결과를 바탕으로 최종 답변 생성)**

* Tool 실행 결과가 담긴 State는 다시 **1단계의 Agent 노드로 전달**됩니다.
* 이제 LLM은 사용자의 원래 질문과 Tool 실행 결과("오늘 서울의 날씨는 맑음, 25도입니다.")를 모두 알고 있는 상태입니다.
* 이 모든 정보를 종합하여 사용자에게 보여줄 최종적인, 자연스러운 답변을 생성합니다.
  + 예: "네, 오늘 서울의 날씨는 맑고 기온은 25도입니다."
* 이 최종 답변은 State에 저장되고, 라우터를 거쳐 END로 전달되어 워크플로우가 종료됩니다.