코드 실습 2

State Graph

- 상태(state)를 기반으로 작동하는 그래프 구조
- 실습 : 레스토랑 메뉴 추천 시스템
  - 사용자의 선호도에 따라 메뉴를 추천하고, 메뉴에 대한 정보 제공
- 상태(state) : 그래프가 처리하는 데이터의 구조를 정의

  기존 상태를 override(덮어쓰기)

```
from typing import TypedDict

# 상태 Schema 정의 - 사용자의 선호도, 추천된 메뉴, 그리고 메뉴 정보를 저장
class MenuState(TypedDict):
    user_preference: str
    recommended_menu: str
    menu_info: str
```

- MenuState라는 클래스 속 객체들을 str 형태로 정의

노드(Node) : 그래프에서 실제 작업을 수행하는 함수

```
import random

def get_user_preference(state: MenuState) → MenuState:
    print("---랜덤 사용자 선호도 생성---")
    preferences = ["육류", "해산물", "채식", "아무거나"]
    preference = random.choice(preferences)
    print(f"생성된 선호도: {preference}")
    return {"user_preference": preference}

def recommend_menu(state: MenuState) → MenuState:
    print("---메뉴 추천---")
    preference = state['user_preference']
```

```
    if preference == "육류":
        menu = "스테이크"
    elif preference == "해산물":
        menu = "랍스터 파스타"
    elif preference == "채식":
        menu = "그린 샐러드"
    else:
        menu = "오늘의 쉐프 특선"
    print(f"추천 메뉴: {menu}")
    return {"recommended_menu": menu}

def provide_menu_info(state: MenuState) → MenuState:
    print("---메뉴 정보 제공---")
    menu = state['recommended_menu']
    if menu == "스테이크":
        info = "최상급 소고기로 만든 juicy한 스테이크입니다. 가격: 30,000원"
    elif menu == "랍스터 파스타":
        info = "신선한 랍스터와 al dente 파스타의 조화. 가격: 28,000원"
    elif menu == "그린 샐러드":
        info = "신선한 유기농 채소로 만든 건강한 샐러드. 가격: 15,000원"
    else:
        info = "쉐프가 그날그날 엄선한 특별 요리입니다. 가격: 35,000원"
    print(f"메뉴 정보: {info}")
    return {"menu_info": info}
```

- 사용자 선호에 따른 분기별 함수 정의

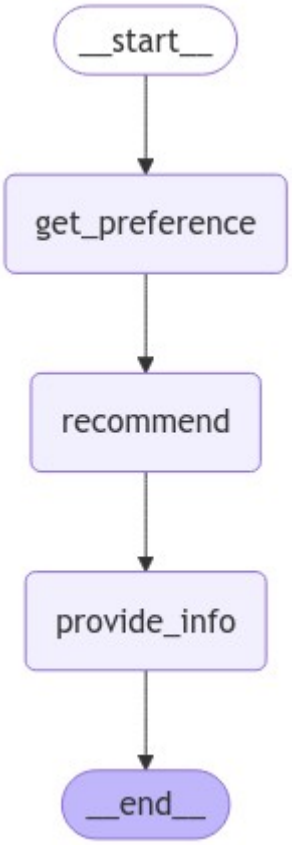
그래프(Graph) 구성

- 정의한 구성 요소들을 사용해 전체 그래프를 빌드

```
from langgraph.graph import StateGraph, START, END

# 그래프 빌더 생성
builder = StateGraph(MenuState)

# 노드 추가
builder.add_node("get_preference", get_user_preference)
builder.add_node("recommend", recommend_menu)
```

```
builder.add_node("provide_info", provide_menu_info)

# 엣지 추가
builder.add_edge(START, "get_preference")
builder.add_edge("get_preference", "recommend")
builder.add_edge("recommend", "provide_info")
builder.add_edge("provide_info", END)

# 그래프 컴파일
graph = builder.compile()
```

- 앞서 구성한 함수들을 연결해 그래프를 구축
- StateGraph 생성
  - builder = StateGraph(MenuState)는 MenuState라는 상태 객체를 관리하는 그래프 빌더를 생성함. 이 상태 객체는 각 노드(함수) 간 데이터를 전달하는 역할
- .add_node : 노드 추가
  - 각각의 함수를 그래프의 노드로 등록함
- .add_edge : 엣지 추가
  - 각 노드를 연결함
- builder.compile : 위에서 정의한 노드와 엣지들을 기반으로 실행 가능한 워크플로우 그래프를 완성
- 최종적으로 사용자 선호도 생성 → 메뉴 추천 → 메뉴 정보 제공의 순서




```
# 그래프 실행

def print_result(result: MenuState):
    print("\n=== 결과 ===")
    print("선호도:", result['user_preference'])
    print("추천 메뉴:", result['recommended_menu'])
    print("메뉴 정보:", result['menu_info'])
    print("============\n")


# 초기 입력
inputs = {"user_preference": ""}

# 여러 번 실행하여 테스트
for _ in range(2):
    result = graph.invoke(inputs)
    print_result(result)
```

```
    print("*"*100)
    print()
```

- 그래프 실행. input 메서드로 사용자 정보를 받아 그래프 실행

조건부 엣지

- 엣지는 노드 간의 연결을 정의함
- 조건부 엣지 : 사용자 입력이 메뉴 관련인지 여부에 따라 다른 경로로 진행

```
from typing import List

# state 스키마
class MenuState(TypedDict):
    user_query: str
    is_menu_related: bool
    search_results: List[str]
    final_answer: str
```

- State 정의
    - 사용자 입력이 메뉴 추천이면 벡터 저장소에서 검색해 RAG Chain을 실행
    - 그렇지 않은 경우 LLM이 답변을 생성

```
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings

embeddings_model = OllamaEmbeddings(model="bge-m3")

# Chroma 인덱스 로드
vector_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="restaurant_menu",
```

```
    persist_directory="./chroma_db",
)
```

- 벡터 저장소 검색 도구
    - 메뉴 검색을 위한 벡터 저장소를 초기화 (기존 저장소를 로드)
- Chroma를 통해 위에서 정의한 embedding 모델을 사용, 데이터 파일명, 경로명 지정

노드 생성

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

# LLM 모델
llm = ChatOpenAI(model="gpt-4o-mini")

def get_user_query(state: MenuState) -> MenuState:
    user_query = input("무엇을 도와드릴까요? ")
    return {"user_query": user_query}

def analyze_input(state: MenuState) -> MenuState:
    analyze_template = """
    사용자의 입력을 분석하여 레스토랑 메뉴 추천이나 음식 정보에 관한 질문인지 판단하세요.

    사용자 입력: {user_query}

    레스토랑 메뉴나 음식 정보에 관한 질문이면 "True", 아니면 "False"로 답변하세요.

    답변:
    """
    analyze_prompt = ChatPromptTemplate.from_template(analyze_template)
    analyze_chain = analyze_prompt | llm | StrOutputParser()

    result = analyze_chain.invoke({"user_query": state['user_query']})
```

```
    is_menu_related = result.strip().lower() == "true"

    return {"is_menu_related": is_menu_related}

def search_menu_info(state: MenuState) -> MenuState:
    # 벡터저장소에서 최대 2개의 문서를 검색
    results = vector_db.similarity_search(state['user_query'], k=2)
    search_results = [doc.page_content for doc in results]
    return {"search_results": search_results}

def generate_menu_response(state: MenuState) -> MenuState:
    response_template = """
    사용자 입력: {user_query}
    메뉴 관련 검색 결과: {search_results}

    위 정보를 바탕으로 사용자의 메뉴 관련 질문에 대한 상세한 답변을 생성하세요.
    검색 결과의 정보를 활용하여 정확하고 유용한 정보를 제공하세요.

    답변:
    """
    response_prompt = ChatPromptTemplate.from_template(response_templ
ate)
    response_chain = response_prompt | llm | StrOutputParser()

    final_answer = response_chain.invoke({"user_query": state['user_quer
y'], "search_results": state['search_results']})
    print(f"\n메뉴 어시스턴트: {final_answer}")

    return {"final_answer": final_answer}

def generate_general_response(state: MenuState) -> MenuState:
    response_template = """
    사용자 입력: {user_query}

    위 입력은 레스토랑 메뉴나 음식과 관련이 없습니다.
    일반적인 대화 맥락에서 적절한 답변을 생성하세요.

    답변:
```

```
    """
    response_prompt = ChatPromptTemplate.from_template(response_templ
ate)
    response_chain = response_prompt | llm | StrOutputParser()

    final_answer = response_chain.invoke({"user_query": state['user_quer
y']})
    print(f"\n일반 어시스턴트: {final_answer}")

    return {"final_answer": final_answer}
```

- 메뉴 관련 질문과 일반 질문을 구분해 다른 방식으로 답변을 생성
- get_user_query(state : MenuState) : 사용자로부터 질문을 입력받는 함수. input 함수로 사용자와 상호작용
- analyze_input(state : MenuState) : 분기점 역할. 사용자의 질문을 LLM에 보내 질문이 메뉴 관련인지 여부를 판단하고, 이를 Bool 타입으로 받음
- search_menu_info(state: MenuState) : 질문이 메뉴와 관련있을 때, 실행. vector_db라는 가상의 벡터 DB에서 사용자의 질문과 가장 유사한 메뉴 정보 탐색. 검색 결과는 RAG 파이프라인에서 추가적인 맥락 정보로 사용
- generate_menu_response : 질문이 메뉴 관련일 때, 최종 답변 생성. 사용자 질문과 search_menu_info에서 검색된 정보를 LLM에게 함께 전달해 답변 생성 명령
- generate_general_response : 질문이 메뉴와 관련없을 때, 실행. 검색 결과 없이 단순하게 대화 맥락으로 LLM에게 답변 생성 요청

엣지

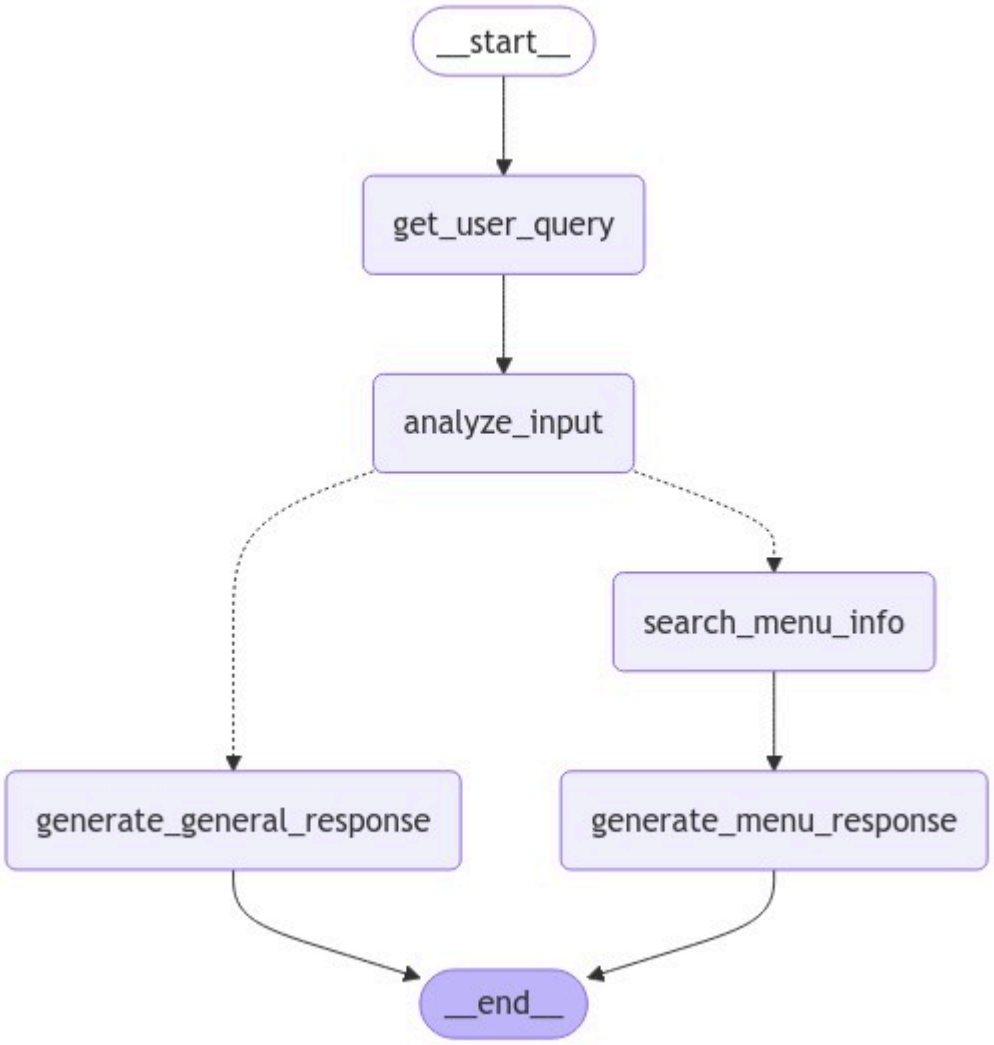
```
from typing import Literal

def decide_next_step(state: MenuState) -> Literal["search_menu_info", "ge
nerate_general_response"]:
    if state['is_menu_related']:
        return "search_menu_info"
    else:
        return "generate_general_response"
```

- 조건부 라우팅 기능. decide_next_step 함수로 이전 함수의 실행결과에 따라 다음 실행 노드 동적으로 결정

그래프

```
from langgraph.graph import StateGraph, START, END

# 그래프 구성
builder = StateGraph(MenuState)

# 노드 추가
builder.add_node("get_user_query", get_user_query)
builder.add_node("analyze_input", analyze_input)
builder.add_node("search_menu_info", search_menu_info)
builder.add_node("generate_menu_response", generate_menu_response)
builder.add_node("generate_general_response", generate_general_respons
e)

# 엣지 추가
builder.add_edge(START, "get_user_query")
builder.add_edge("get_user_query", "analyze_input")

# 조건부 엣지 추가
builder.add_conditional_edges(
    "analyze_input",
    decide_next_step,
    {
        "search_menu_info": "search_menu_info",
        "generate_general_response": "generate_general_response"
    }
)

builder.add_edge("search_menu_info", "generate_menu_response")
builder.add_edge("generate_menu_response", END)
builder.add_edge("generate_general_response", END)
```

```
# 그래프 컴파일
graph = builder.compile()
```




```
while True:
    initial_state = {'user_query':''}
    graph.invoke(initial_state)
    continue_chat = input("다른 질문이 있으신가요? (y/n): ").lower()
    if continue_chat != 'y':
        print("대화를 종료합니다. 감사합니다!")
        break
```

- 사용자가 직접 종료하기 전까지 루프를 실행해 대화형 루프 구현