코드 실습 1

```
from langchain_openai import ChatOpenAI

# ChatOpenAI 모델 초기화
llm = ChatOpenAI(model="gpt-4o-mini", max_tokens = 100)

# 웹 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=[web_search])
```

- 여기서 web search는 유료 플랜에서만 가능하기에 할당량 초과 오류가 발생.

invoke 메서드

- llm에게 프롬프트 input과 같이 입력을 주고, 출력을 받음

```
tool_call = ai_msg.tool_calls[0]
tool_call
```

- tool_calls 메서드는 llm 모델의 output 뭉치를 말하며, 일반적으로 dictionary, list 형태로 슬라이싱 가능

```
from langchain_core.messages import ToolMessage
tool_message = ToolMessage(
    content=tool_output,
    tool_call_id=tool_call["id"],
    name=tool_call["name"]
)

print(tool_message)
```

- ToolMessage : 응답 결과를 다시 LLM에게 넘겨줌.
    - content = tool_output : API를 호출해서 얻은 실제 결과값

- tool_call_id = tool_call[”id”] : LLM을 이전에 호출했을 때, 붙여진 고유한 ID. 이걸 알려줘야 LLM이 전에 대답한 것의 결과라는 것을 인식할 수 있음. 즉, 결과를 다시 알려줄 때, 어떤 ID의 응답에 대한 내용인지를 알려줌
- name = tool_call[”name”] : LLM이 호출을 요청했던 도구의 이름

```
### 방법 3: 도구 직접 호출하여 바로 ToolMessage 객체 생성

# 이 방법은 도구를 직접 호출하여 ToolMessage 객체를 생성한다.
# 가장 간단하고 직관적인 방법으로, LangChain의 추상화를 활용한다.

tool_message = web_search.invoke(tool_call)

print(tool_message)
```

- 개발자가 직접 요청/응답 객체를 생성할 필요 없이 LLM과 같은 도구에게 요청 및 결과를 받을 수 있게 함

```
from datetime import datetime
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableConfig, chain

# 오늘 날짜 설정
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
prompt = ChatPromptTemplate([
    ("system", f"You are a helpful AI assistant. Today's date is {today}."),
    ("human", "{user_input}"),
    ("placeholder", "{messages}"),
])

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

# LLM에 도구를 바인딩
llm_with_tools = llm.bind_tools(tools=[web_search])

# LLM 체인 생성
```

```
llm_chain = prompt | llm_with_tools

# 도구 실행 체인 정의
@chain
def web_search_chain(user_input: str, config: RunnableConfig):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config=config)
    print("ai_msg: \n", ai_msg)
    print("-"*100)
    tool_msgs = web_search.batch(ai_msg.tool_calls, config=config)
    print("tool_msgs: \n", tool_msgs)
    print("-"*100)
    return llm_chain.invoke({**input_, "messages": [ai_msg, *tool_msgs]}, co
nfig=config)

# 체인 실행
response = web_search_chain.invoke("오늘 모엣샹동 샴페인의 가격은 얼마인가
요?")

# 응답 출력
pprint(response.content)
```

- llm_chain으로 첫 번째 체인을 생성. 이 체인은 첫 번째 LLM 호출을 담당.
- @chain을 사용해 전체 흐름을 관리하는 체인 생성.
- LLM에게 질문 → LLM이 web_search를 통해 검색 → 검색 결과를 LLM이 구체적인 답변을 생성해 반환

```
from langchain_community.tools import TavilySearchResults
from langchain_core.tools import tool
from typing import List

# Tool 정의
@tool
def search_web(query: str) → str:
    """Searches the internet for information that does not exist in the databa
se or for the latest information."""
```

```
    tavily_search = TavilySearchResults(max_results=2)
    docs = tavily_search.invoke(query)

    formatted_docs = "\n---\n".join([
        f'<Document href="{doc["url"]}"/>\n{doc["content"]}\n</Document>'
        for doc in docs
        ])

    if len(formatted_docs) > 0:
        return formatted_docs

    return "관련 정보를 찾을 수 없습니다."
```

- 데코레이터를 사용한 tool 정의 예시

Langchain_core 기능 활용하기 (문서 분할)

```
from langchain_core.documents import Document

# 문서 분할 (Chunking)
def split_menu_items(document):
    """
    메뉴 항목을 분리하는 함수
    """
    # 정규표현식 정의
    pattern = r'(\d+\.\s.*?)(?=\n\n\d+\.|$)'
    menu_items = re.findall(pattern, document.page_content, re.DOTALL)

    # 각 메뉴 항목을 Document 객체로 변환
    menu_documents = []
    for i, item in enumerate(menu_items, 1):
        # 메뉴 이름 추출
        menu_name = item.split('\n')[0].split('.', 1)[1].strip()

        # 새로운 Document 객체 생성
        menu_doc = Document(
            page_content=item.strip(),
```

```
            metadata={
                "source": document.metadata['source'],
                "menu_number": i,
                "menu_name": menu_name
            }
        )
        menu_documents.append(menu_doc)

    return menu_documents


# 메뉴 항목 분리 실행
menu_documents = []
for doc in documents:
    menu_documents += split_menu_items(doc)

# 결과 출력
print(f"총 {len(menu_documents)}개의 메뉴 항목이 처리되었습니다.")
for doc in menu_documents[:2]:
    print(f"\n메뉴 번호: {doc.metadata['menu_number']}")
    print(f"메뉴 이름: {doc.metadata['menu_name']}")
    print(f"내용:\n{doc.page_content[:100]}...")
```

- pattern을 통해 정규 표현식을 정의
- re.findall(pattern, string, flags(optional))
  - pattern : 찾고자 하는 패턴을 정의
  - string : 패턴을 적용할 원본 문자열
  - flags(선택사항) : 정규 표현식의 동작 방식을 변경하는 옵션. 여러 옵션을 함께 사용 가능
- re.DOTALL : .의 메타 문자의 의미를 변경. re.DOTALL이 있으면 줄바꿈 문자를 포함한 모든 문자와 일치해야함. 현재 코드에서는 menu_items가 여러 줄에 걸쳐 있기 때문에 re.DOTALL 사용
- 분리된 단순 문자열의 document를 LangChain이 인식할 수 있는 Document 객체로 만듦

- 메타 데이터 정의를 통해 metadata라는 딕셔너리 객체에 source, menu_number, menu_name과 같은 유용한 정보를 저장. 이를 통해 LLM은 구조화된 정보를 함께 이해할 수 있게 됨

```
# Chroma Vectorstore를 사용하기 위한 준비
from langchain_chroma import Chroma
from langchain_ollama  import OllamaEmbeddings

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

# Chroma 인덱스 생성
menu_db = Chroma.from_documents(
    documents=menu_documents,
    embedding=embeddings_model,
    collection_name="restaurant_menu",
    persist_directory="./chroma_db",
)

# Retriever 생성
menu_retriever = menu_db.as_retriever(
    search_kwargs={'k': 2},
)

# 쿼리 테스트
query = "시그니처 스테이크의 가격과 특징은 무엇인가요?"
docs = menu_retriever.invoke(query)
print(f"검색 결과: {len(docs)}개")

for doc in docs:
    print(f"메뉴 번호: {doc.metadata['menu_number']}")
    print(f"메뉴 이름: {doc.metadata['menu_name']}")
    print()
```

단계별 정리

1. 임베딩 모델 준비 및 Chroma 인덱스 생성.
    - embedding model을 bge-m3라는 모델로 설정
    - 이전에 분할했던 document를 menu_db로 받음

    - documents : 벡터화할 문서들
    - embedding : 문서들을 벡터로 변환하는 데 사용할 임베딩 모델 정의
    - collection_name : 데이터베이스 내 컬렉션의 이름
    - persist_directory : 벡터 데이터베이스를 저장할 로컬 디렉토리 정의
    - 이 과정을 통해 menu_documents의 내용이 벡터로 변환되고, 이 벡터들은 Chroma 컬렉션에 저장됨
2. Retriever 정의
    - 생성된 Chroma 데이터베이스에서 문서를 검색할 수 있는 Retriever 객체를 만듦
    - search_kwargs는 검색 쿼리와 가장 유사한 문서를 몇 개 가져올건지 정의
3. query 테스트
    - 이전 search_kwargs를 통해 2개의 유사 문서를 가져옴
    - menu_retriever.invoke(query)를 통해 Retriever를 호출해 질문 (query)와 가장 유사한 문서 2개를 DB에서 검색
    - Retriever는 질문을 임베딩 모델로 벡터화 후, DB에 저장된 문서 벡터들과의 유사도를 비교해 가장 유사한 문서 반환
    - 반환값은 Document 객체의 리스트 형태

- 결과적으로 위 코드는 LLM이 최신 정보나 내부 데이터를 활용할 수 있게 외부 지식을 검색하는 시스템을 구축함.
- 이는 LLM이 기존 학습한 정보 외에 DB에 있는 구체적인 정보를 바탕으로 더 정확하고 풍부한 답변을 생성할 수 있게 함.

```
# 벡터 저장소 로드
menu_db = Chroma(
    embedding_function=embeddings_model,
    collection_name="restaurant_menu",
    persist_directory="./chroma_db",
)

@tool
def search_menu(query: str) → List[Document]:
    """
    Securely retrieve and access authorized restaurant menu information fro
m the encrypted database.
```

```
    Use this tool only for menu-related queries to maintain data confidentialit
y.
    """
    docs = menu_db.similarity_search(query, k=2)
    if len(docs) > 0:
        return docs

    return [Document(page_content="관련 메뉴 정보를 찾을 수 없습니다.")]

# 도구 속성
print("자료형: ")
print(type(search_menu))
print("-"*100)

print("name: ")
print(search_menu.name)
print("-"*100)

print("description: ")
pprint(search_menu.description)
print("-"*100)

print("schema: ")
pprint(search_menu.args_schema.schema())
print("-"*100)
```

- menu_db라는 객체에 이전 단계에서 persist_directory에 저장해둔 Chroma 데이터베이스를 다시 메모리로 불러옴. 이미 정의된 db이기 때문에 from_documents와는 다르게 작동
- embedding_function : 벡터 유사도 검색을 위해 이전과 동일한 임베딩 모델을 지정. 쿼리를 올바른 벡터로 변환해 DB의 벡터들과 유사도 비교를 원활히 수행하기 위함
- collection_name, persist_directory를 지정. 저장한 컬렉션 이름과 경로를 사용
- @tool 데코레이터를 통해 사용자 tool 정의 후, 유사도 검색으로 상위 2개의 검색 결과를 가져옴. 이 결과가 0이하일 경우에는 메시지 출력

```
# LLM에 도구를 바인딩 (2개의 도구 바인딩)
llm_with_tools = llm.bind_tools(tools=[search_menu, search_wine])

# 도구 호출이 필요한 LLM 호출을 수행
query = "시그니처 스테이크의 가격과 특징은 무엇인가요? 그리고 스테이크와 어울리는 와인 추천도 해주세요."
ai_msg = llm_with_tools.invoke(query)

# LLM의 전체 출력 결과 출력
pprint(ai_msg)
print("-" * 100)

# 메시지 content 속성 (텍스트 출력)
pprint(ai_msg.content)
print("-" * 100)

# LLM이 호출한 도구 정보 출력
pprint(ai_msg.tool_calls)
print("-" * 100)
```

- LLM 모델이 search_menu와 search_wine 두 가지 도구를 사용할 수 있게 연결
- query에 복합 질문을 주면서 호출
- 이 과정에서 LLM은 스테이크 메뉴 관련 호출에 search_menu를 호출해 정보를 탐색하고 와인 관련 호출에 search_wine을 호출해 정보를 탐색해 추론 결과를 바탕으로 사용자 질문에 답변을 주는 대신 ToolMessage를 생성해 반환
- LLM의 multi-tool call 능력을 보여줌

```
tools = [search_web, wiki_summary, search_wine, search_menu]
for tool in tools:
    print(tool.name)
```

- 복합 답변 생성을 위해 여러 tool을 연결

```
from datetime import datetime
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableConfig, chain

# 오늘 날짜 설정
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
prompt = ChatPromptTemplate([
    ("system", f"You are a helpful AI assistant. Today's date is {today}."),
    ("human", "{user_input}"),
    ("placeholder", "{messages}"),
])

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

# 4개의 검색 도구를 LLM에 바인딩
llm_with_tools = llm.bind_tools(tools=tools)

# LLM 체인 생성
llm_chain = prompt | llm_with_tools

# 도구 실행 체인 정의
@chain
def restaurant_menu_chain(user_input: str, config: RunnableConfig):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config=config)

    tool_msgs = []
    for tool_call in ai_msg.tool_calls:
        print(f"{tool_call['name']}: \n{tool_call}")
        print("-"*100)

        if tool_call["name"] == "search_web":
            tool_message = search_web.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)
```

```
        elif tool_call["name"] == "wiki_summary":
            tool_message = wiki_summary.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_wine":
            tool_message = search_wine.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_menu":
            tool_message = search_menu.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)

    print("tool_msgs: \n", tool_msgs)
    print("-"*100)
    return llm_chain.invoke({**input_, "messages": [ai_msg, *tool_msgs]}, co
nfig=config)

# 체인 실행
response = restaurant_menu_chain.invoke("시그니처 스테이크의 가격과 특징은
무엇인가요? 그리고 스테이크와 어울리는 와인 추천도 해주세요.")

# 응답 출력
print(response.content)
```

- 현재 날짜를 포함해 사용자 프롬프트 템플릿 정의.
- 모델 초기화 진행
- llm_with_tools : 이전 단계에서 복합 데이터베이스를 연결한 tools를 LLM 모델에 바인딩 진행
- LLM에 query 입력 → LLM의 도구 호출 → 루프를 실행 → name을 기준으로 도구 호출 → 각 도구들의 ToolMessage 생성 → tool_msgs에 저장 → LLM 재호출 (1차 호출에서 반환한 user_input, messages를 포함한 LLM) → 2차 호출의 LLM은 답변 생성 시, 1차 호출에서 반환된 message를 포함한 답변을 사용자에게 전달

few-shot 프롬프팅

```
from langchain_core.messages import AIMessage, HumanMessage, ToolMessage
from langchain_core.prompts import ChatPromptTemplate

examples = [
    HumanMessage("트러플 리조또의 가격과 특징, 그리고 어울리는 와인에 대해 알려주세요.", name="example_user"),
    AIMessage("메뉴 정보를 검색하고, 위키피디아에서 추가 정보를 찾은 후, 어울리는 와인을 검색해보겠습니다.", name="example_assistant"),
    AIMessage("", name="example_assistant", tool_calls=[{"name": "search_menu", "args": {"query": "트러플 리조또"}, "id": "1"}]),
    ToolMessage("트러플 리조또: 가격 ₩28,000, 이탈리아 카나롤리 쌀 사용, 블랙 트러플 향과 파르메산 치즈를 듬뿍 넣어 조리", tool_call_id="1"),
    AIMessage("트러플 리조또의 가격은 ₩28,000이며, 이탈리아 카나롤리 쌀을 사용하고 블랙 트러플 향과 파르메산 치즈를 듬뿍 넣어 조리합니다. 이제 추가 정보를 위키피디아에서 찾아보겠습니다.", name="example_assistant"),
    AIMessage("", name="example_assistant", tool_calls=[{"name": "wiki_summary", "args": {"query": "트러플 리조또", "k": 1}, "id": "2"}]),
    ToolMessage("트러플 리조또는 이탈리아 요리의 대표적인 리조또 요리 중 하나로, 고급 식재료인 트러플을 사용하여 만든 크리미한 쌀 요리입니다. 주로 아르보리오나 카나롤리 등의 쌀을 사용하며, 트러플 오일이나 생 트러플을 넣어 조리합니다. 리조또 특유의 크리미한 질감과 트러플의 강렬하고 독특한 향이 조화를 이루는 것이 특징입니다.", tool_call_id="2"),
    AIMessage("트러플 리조또의 특징에 대해 알아보았습니다. 이제 어울리는 와인을 검색해보겠습니다.", name="example_assistant"),
    AIMessage("", name="example_assistant", tool_calls=[{"name": "search_wine", "args": {"query": "트러플 리조또에 어울리는 와인"}, "id": "3"}]),
    ToolMessage("트러플 리조또와 잘 어울리는 와인으로는 주로 중간 바디의 화이트 와인이 추천됩니다. 1. 샤르도네: 버터와 오크향이 트러플의 풍미를 보완합니다. 2. 피노 그리지오: 산뜻한 산미가 리조또의 크리미함과 균형을 이룹니다. 3. 베르나차: 이탈리아 토스카나 지방의 화이트 와인으로, 미네랄리티가 트러플과 잘 어울립니다.", tool_call_id="3"),
    AIMessage("트러플 리조또(₩28,000)는 이탈리아의 대표적인 리조또 요리 중 하나로, 이탈리아 카나롤리 쌀을 사용하고 블랙 트러플 향과 파르메산 치즈를 듬뿍 넣어 조리합니다. 주요 특징으로는 크리미한 질감과 트러플의 강렬하고 독특한 향이 조화를 이루는 점입니다. 고급 식재료인 트러플을 사용해 풍부한 맛과 향을 내며, 주로 아르보리오나 카나롤리 등의 쌀을 사용합니다. 트러플 리조또와 잘 어울리는 와인으로는 중간 바디의 화이트 와인이 추천됩니다. 특히 버터와 오크향이 트러플의 풍미를 보완하는 샤
```

```
르도네, 산뜻한 산미로 리조또의 크리미함과 균형을 이루는 피노 그리지오, 그리고 미네랄리티가 트러플과 잘 어울리는 이탈리아 토스카나 지방의 베르나차 등이 좋은 선택이 될 수 있습니다.", name="example_assistant"),
]

system = """You are an AI assistant providing restaurant menu information and general food-related knowledge.
For information about the restaurant's menu, use the search_menu tool.
For other general information, use the wiki_summary tool.
For wine recommendations or pairing information, use the search_wine tool.
If additional web searches are needed or for the most up-to-date information, use the search_web tool.
"""

few_shot_prompt = ChatPromptTemplate.from_messages([
    ("system", system),
    *examples,
    ("human", "{query}"),
])

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

# 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=tools)

# Few-shot 프롬프트를 사용한 체인 구성
fewshot_search_chain = few_shot_prompt | llm_with_tools

# 체인 실행
query = "스테이크 메뉴가 있나요? 스테이크와 어울리는 와인을 추천해주세요."
response = fewshot_search_chain.invoke(query)

# 결과 출력
for tool_call in response.tool_calls:
    print(tool_call)
```

- example을 참고한 LLM 모델 프롬프팅
- 스테이크 메뉴와 어울리는 와인 추천이 query로 들어올 경우, LLM은 두 가지 행동을 요구받음
- search_menu에서 메뉴 정보를 검색하고, search_wine으로 와인 추천을 검색함
- 즉, LLM에게 단순 도구 활용 능력뿐만 아니라 언제, 어떻게, 어떤 순서로 도구들을 사용해야 하는지에 대한 전략을 교육함.
- 행동 지침을 입력해주면서 LLM의 연산과정을 최소화함
- ChatPromptTemplete을 통해 system으로 정의된 역할을 부여, example을 통해 답변 템플릿 제공, human을 통해 사용자 입력 정의

출력 결과

```
[
   {
      "name" : "search_menu",
      "args" : {
         "query" : "스테이크 메뉴"
         }
      },
      {
         "name" : "search_wine",
         "args" : {
            "query" : "스테이크와 어울리는 와인"
         }
      }
   ]
```

```
# 체인 실행
query = "파스타의 유래에 대해서 알고 있나요? 서울 강남의 파스타 맛집을 추천해주세요."
response = fewshot_search_chain.invoke(query)
```

```
# 결과 출력
for tool_call in response.tool_calls:
    print(tool_call)
```

- 파스타의 유래에는 wiki_summary 도구, 서울 강남의 파스타 맛집에는 search_web 도구를 호출

LLM이 지시한 도구 호출 정보

```
[
   {
      "name" : "wiki_summary",
      "args" : {
         "query" : "파스타의 유래",
         "k" : 1
      }
   },
   {
      "name" : "search_web",
      "args" : {
         "query" : "서울 강남의 파스타 맛집"
      }
   }
]
```

```
from datetime import datetime
from langchain_core.messages import AIMessage, HumanMessage, ToolM
essage
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableConfig, chain
from langchain_openai import ChatOpenAI

# 오늘 날짜 설정
```

```
today = datetime.today().strftime("%Y-%m-%d")

# 프롬프트 템플릿
system = """You are an AI assistant providing restaurant menu information
and general food-related knowledge.
For information about the restaurant's menu, use the search_menu tool.
For other general information, use the wiki_summary tool.
For wine recommendations or pairing information, use the search_wine too
l.
If additional web searches are needed or for the most up-to-date informati
on, use the search_web tool.
"""

few_shot_prompt = ChatPromptTemplate.from_messages([
    ("system", system + f"Today's date is {today}."),
    *examples,
    ("human", "{user_input}"),
    ("placeholder", "{messages}"),
])

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

# 검색 도구를 직접 LLM에 바인딩 가능
llm_with_tools = llm.bind_tools(tools=tools)

# Few-shot 프롬프트를 사용한 체인 구성
fewshot_search_chain = few_shot_prompt | llm_with_tools

# 도구 실행 체인 정의
@chain
def restaurant_menu_chain(user_input: str, config: RunnableConfig):
    input_ = {"user_input": user_input}
    ai_msg = llm_chain.invoke(input_, config=config)

    tool_msgs = []
    for tool_call in ai_msg.tool_calls:
        print(f"{tool_call['name']}: \n{tool_call}")
```

```
        print("-"*100)

        if tool_call["name"] == "search_web":
            tool_message = search_web.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "wiki_summary":
            tool_message = wiki_summary.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_wine":
            tool_message = search_wine.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)

        elif tool_call["name"] == "search_menu":
            tool_message = search_menu.invoke(tool_call, config=config)
            tool_msgs.append(tool_message)

    print("tool_msgs: \n", tool_msgs)
    print("-"*100)
    return fewshot_search_chain.invoke({**input_, "messages": [ai_msg, *to
ol_msgs]}, config=config)


# 체인 실행
query = "스테이크 메뉴가 있나요? 스테이크와 어울리는 와인을 추천해주세요."
response = restaurant_menu_chain.invoke(query)

# 응답 출력
pprint(response.content)
```

- @chain 데코레이터로 사용자 정의 chain 생성
- 1차 LLM 호출로 llm_chain.invoke(input_)을 통해 LLM에게 질문 → LLM은 few-shot을 참고해 search_menu와 search_wine의 두 도구가 필요하다고 판단
- LLM의 도구 호출(AIMessage) 객체 반환, 이 객체는 ai_msg.tool_calls에 저장
- for tool_call in ai_msg.tool_calls 루프 실행
    - search_menu.invoke(...) : 스테이크 메뉴에 대한 정보 탐색

    - search_wine.invoke(...) : 스테이크와 어울리는 와인에 대한 정보 탐색
    - 각 도구 호출 결과는 ToolMessage 객체로 변환해 tool_msgs에 저장
- 2차 LLM 호출 : 모든 도구 탐색 결과가 담긴 메시지를 새로운 입력으로 LLM에 보냄
    - 사용자 원래 질문
    - LLM의 첫 번째 응답 (도구 호출)
    - search_menu, search_wine의 실행 결과
- 위 내용을 모두 담은 LLM의 최종 답변 결과 반환

LangChain에 Agent 사용

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

agent_prompt = ChatPromptTemplate.from_messages([
    ("system", dedent("""
        You are an AI assistant providing restaurant menu information and general food-related knowledge.
        Your main goal is to provide accurate information and effective recommendations to users.

        Key guidelines:
        1. For restaurant menu information, use the search_menu tool. This tool provides details on menu items, including prices, ingredients, and cooking methods.
        2. For general food information, history, and cultural background, utilize the wiki_summary tool.
        3. For wine recommendations or food and wine pairing information, use the search_wine tool.
        4. If additional web searches are needed or for the most up-to-date information, use the search_web tool.
        5. Provide clear and concise responses based on the search results.
        6. If a question is ambiguous or lacks necessary information, politely ask for clarification.
        7. Always maintain a helpful and professional tone.
```

```
        8. When providing menu information, describe in the order of price, m
ain ingredients, and distinctive cooking methods.
        9. When making recommendations, briefly explain the reasons.
        10. Maintain a conversational, chatbot-like style in your final response
s. Be friendly, engaging, and natural in your communication.



        Remember, understand the purpose of each tool accurately and use th
em in appropriate situations.
        Combine the tools to provide the most comprehensive and accurate a
nswers to user queries.
        Always strive to provide the most current and accurate information.
        """)),
    MessagesPlaceholder(variable_name="chat_history", optional=True),
    ("human", "{input}"),
    MessagesPlaceholder(variable_name="agent_scratchpad"),
])
```

- system 인자에 LLM이 맡을 역할 정의 및 도구 사용 지침 제공
- keyguide에 있는 내용을 통해 어떤 때에 어떤 도구를 사용할지 알려줌
- MessagesPlaceholder
    - variable_name = “chat_history” 인자를 넘겨주면서 이전 대화 내용을 기억하도록 지시
    - variable_name = “agent_scratchpad” 인자를 넘겨주면서 LLM이 자신이 호출한 도구와 그 결과를 기록하고, 다음 행동을 결정하는 데 활용할 수 있게 함

```
# Tool calling Agent 생성
from langchain.agents import AgentExecutor, create_tool_calling_agent

tools = [search_web, wiki_summary, search_wine, search_menu]
agent = create_tool_calling_agent(llm, tools, agent_prompt)

# AgentExecutor 생성
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

- create_tool_calling_agent를 통해 llm, tools, agent_prompt를 결합한 에이전트 만듦
- agent_executor를 통해 호출 시, 도구를 실제로 실행하고, 그 결과를 다시 에이전트에 전달해 다음 행동을 결정하게 함.
    - verbose = True : 에이전트의 모든 내부 동작을 상세하게 터미널에 출력해 개발자가 동작을 쉽게 이해하고 디버깅을 도움

```
# AgentExecutor 실행

query = "시그니처 스테이크의 가격과 특징은 무엇인가요? 그리고 스테이크와 어울리는 와인 추천도 해주세요."
agent_response = agent_executor.invoke({"input": query})

pprint(agent_response)
```

Gradio 활용

```
import gradio as gr
from typing import List, Tuple

def answer_invoke(message: str, history: List[Tuple[str, str]]) → str:
    try:
        # 채팅 기록을 AI에게 전달할 수 있는 형식으로 변환
        chat_history = []
        for human, ai in history:
            chat_history.append(HumanMessage(content=human))
            chat_history.append(AIMessage(content=ai))

        # agent_executor를 사용하여 응답 생성
        response = agent_executor.invoke({
            "input": message,
            "chat_history": chat_history[-2:]    # 최근 2개의 메시지 기록만을 활용
        })
```

```
        # agent_executor의 응답에서 최종 답변 추출
        return response['output']
    except Exception as e:
        # 오류 발생 시 사용자에게 알리고 로그 기록
        print(f"Error occurred: {str(e)}")
        return "죄송합니다. 응답을 생성하는 동안 오류가 발생했습니다. 다시 시도해 주세요."

# 예제 질문 정의
example_questions = [
    "시그니처 스테이크의 가격과 특징을 알려주세요.",
    "트러플 리조또와 잘 어울리는 와인을 추천해주세요.",
    "해산물 파스타의 주요 재료는 무엇인가요? 서울 강남 지역에 레스토랑을 추천해주세요.",
    "채식주의자를 위한 메뉴 옵션이 있나요?"
]

# Gradio 인터페이스 생성
demo = gr.ChatInterface(
    fn=answer_invoke,
    title="레스토랑 메뉴 AI 어시스턴트",
    description="메뉴 정보, 추천, 음식 관련 질문에 답변해 드립니다.",
    examples=example_questions,
    theme=gr.themes.Soft()
)

# 데모 실행
demo.launch()
```

- Gradio : 머신러닝 모델이나 데이터 과학 프로젝트를 위한 웹 기반 사용자 인터페이스를 만듦
- answer_invoke
    - 입력 : Gradio 채팅창에 사용자가 입력한 메시지(message)와 이전 대화 기록(history)을 받음
    - 대화 기록 변환 : Gradio의 대화 기록 형식을 LangChain의 HumanMessage와 AIMessage객체로 변환

    - 에이전트 실행 : agent_executor.invoke를 호출해 사용자 입력과 함께 변환된 대화 기록을 에이전트에 전달
        - chat_history[-2:]를 통해 마지막 대화 1턴만 기억하도록 설정
    - 응답 반환 : 에이전트 최종 응답에서 response['output']을 추출해 Gradio 화면에 표시. 오류 발생 시, 오류 메시지 출력
- gr.ChatInterface는 answer_invoke 함수를 통해 최종적으로 보게 될 챗봇 인터페이스를 만듦
    - fn = answer_invoke를 통해 메시지가 입력될 때마다 answer_invoke 함수 실행
    - title, description을 통해 제목과 내용 설정
    - examples : 예제 정의해 agent의 역할 지정
    - demo.launch() : 로컬 웹 서버 시작
    - theme=gr.themes.Soft() : Gradio 인터페이스의 색 테마 지정