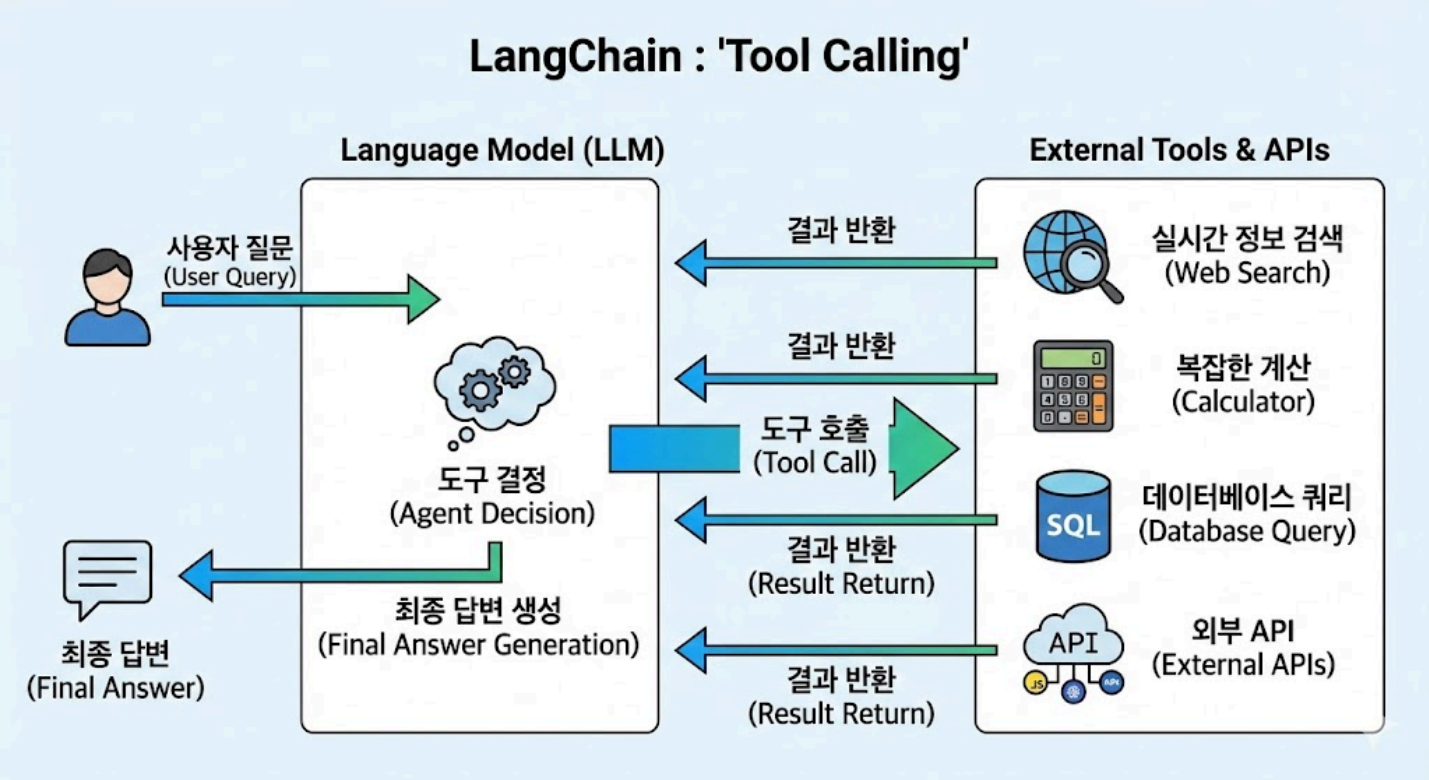
[13주차] 도구 (TOOL)

1. 도구 호출(Tool Calling)이란?

도구 호출은 **언어 모델(LLM)이 외부 기능이나 API를 사용할 수 있게 해주는 핵심 기능**입니다.

- **목적:** AI 모델이 단순한 텍스트 생성을 넘어, **실시간 정보 검색, 복잡한 계산, 데이터베이스 쿼리** 등 실제 작업을 수행할 수 있도록 능력을 확장합니다.
- **중요성:** 특히, 여러 단계의 계획과 실행이 필요한 **에이전트(Agent) 기반 시스템**을 구축하는 데 필수적입니다.
  - **흐름:** 모델이 '질문'을 받고 -> '어떤 도구를 써야 할지' 결정 -> 도구 호출 -> 도구 실행 결과 획득 -> 결과를 바탕으로 최종 답변 생성.



2. LangChain의 내장(Pre-built) 도구 구조

도구(Tool)와 툴킷(Toolkit)의 정의

- **도구(Tool):** 모델이 호출하도록 설계된 **단일 유틸리티**입니다.
  - **입력:** 모델이 생성하도록 설계됩니다.
  - **출력:** 실행 후 모델에게 다시 전달되어 최종 응답에 활용됩니다.
- **툴킷(Toolkit):** 함께 사용되도록 의도된 **도구들의 모음**입니다. (예: `Gmail Toolkit` 은 이메일 읽기, 쓰기 등 여러 도구를 포함)

LangChain은 다양한 작업을 자동화하고 외부 데이터 소스와 상호작용하기 위한 풍부한 내장 도구 세트를 제공합니다.

주요 도구 카테고리 및 활용

LangChain의 내장 도구들은 기능에 따라 다음과 같이 분류됩니다.

카테고리	주요 역할	예시 도구	참고 사항
1. 검색 도구	웹 검색 및 실시간 정보 획득	**Tavily Search** (월 1,000회 무료), Brave Search (무료), Google Search, Bing Search	URL, 스니펫, 제목 등의 데이터를 반환
2. 코드 인터프리터	코드 실행 환경 제공	Azure Container Apps, E2B Data Analysis	Python, JavaScript 등 실행. 텍스트/이미지 등 결과를 반환
3. 생산성 도구	외부 생산성 툴 작업 자동화	**Github Toolkit**, **Gmail Toolkit**, Slack Toolkit	대부분 무료이나, API 사용량 제한이 있을 수 있음
4. 웹 브라우징 도구	웹 브라우저 상호작용/HTTP 요청	PlayWright Browser Toolkit, Requests Toolkit	브라우저 자동화 또는 HTTP 요청 수행
5. 데이터베이스 도구	데이터베이스 작업 자동화	**SQLDatabase Toolkit**, Spark SQL Toolkit	SQL 쿼리 실행 등 데이터베이스 작업을 수행

*스니펫(snippet)은 **재사용 가능한 소스 코드, 기계어, 텍스트의 작은 부분**을 일컫는 프로그래밍 용어

LangChain 도구의 가치

LangChain의 도구들은 언어 모델의 기능을 **획기적으로 확장**시켜 줍니다.

개발자는 이 도구들을 활용하여 단순한 챗봇을 넘어, **외부 서비스 및 리소스와 상호작용**하며 실세계의 문제를 해결하는 **더욱 강력하고 다양한 AI 애플리케이션**을 구축할 수 있습니다.

전체 지원 도

https://docs.langchain.com/oss/javascript/integrations/tools

웹 검색 도구 활용

1. Tavily Search

Tavily Search API는 LLM과 RAG 시스템에 최적화된 검색 엔진입니다. 기존의 일반 검색 엔진들과 달리, Tavily는 AI 애플리케이션의 요구사항에 맞춰 설계되었습니다.

Tavily의 주요 특징

- AI 최적화: LLM과의 원활한 통합을 위해 설계되었습니다.
- 최신 정보: 실시간으로 업데이트 되는 정보를 제공합니다.

- 다양한 검색 모드: 일반 검색, 뉴스 검색, 이미지 검색 등 다양한 모드를 제공합니다.
- 콘텐츠 필터링: 신뢰할 수 있는 소스의 정보 만을 제공합니다.
- 무료 사용량: 월 1,000회의 무료 API 호출을 제공합니다.

Tavily Search의 작동 원리

Tavily Search는 자체적으로 웹을 크롤링하는 전통적인 검색 엔진과는 다르게, **여러 검색 엔진 및 데이터 소스의 결과를 통합하여 최종적으로 최적화된 응답을 제공하는 에이전트 기반의 검색 API**입니다. 이는 특히 LLM(대규모 언어 모델)이 최신 정보에 접근하고 사실 확인을 수행하는 RAG(검색 증강 생성) 워크플로우에 초점을 맞추고 있습니다.

Tavily Search가 내부적으로 동작하는 주요 단계는 다음과 같습니다.

---

1. 쿼리 최적화 및 확장 (Query Optimization)

사용자나 LLM이 최초 쿼리(질문)를 입력하면, Tavily는 이 쿼리를 **최적화**하고 **확장**합니다.

- **쿼리 개서 (Query Rewriting):** 입력된 쿼리를 이해하고, 핵심 의도를 파악하여 여러 개의 다양한 검색 엔진 친화적인 쿼리로 변형합니다. 예를 들어, 문맥에 따라 긴 질문을 키워드 위주로 간결하게 바꾸거나, 검색에 필요한 추가적인 정보를 추론하여 쿼리에 삽입합니다.
- **다중 검색어 생성:** 단일 질문에 대해 여러 각도에서 정보를 수집할 수 있도록 다양한 검색어를 생성합니다.

---

2. 다중 소스 데이터 수집 (Multi-Source Aggregation)

Tavily는 자체적인 인덱싱 없이, **다양한 고품질 검색 엔진 및 데이터 소스**에 최적화된 쿼리를 동시에 전송합니다.

- **API 통합:** Tavily는 구글(Google), Bing, 기타 전문 데이터베이스 등 여러 주요 검색 엔진의 API를 통해 결과를 수집합니다.
- **실시간 데이터:** 이 방법을 통해 최신 정보에 대한 접근성이 높습니다.
- **필터링 및 정규화:** 수집된 원시 결과는 Tavily의 시스템 내에서 중복을 제거하고 표준 형식으로 정규화됩니다.

---

3. 결과 랭킹 및 큐레이션 (Ranking & Curation)

수집된 방대한 정보는 Tavily의 핵심 AI 기술을 통해 **가장 관련성이 높고 신뢰할 수 있는 정보**를 선별하는 과정을 거칩니다.

- **AI 기반 랭킹:** Tavily는 기계 학습 모델을 사용하여 각 소스 결과의 품질, 신뢰도, 그리고 원래 쿼리와의 관련성을 평가하고 순위를 매깁니다.
- **스니펫 생성:** 단순히 링크만 제공하는 것이 아니라, 각 링크에서 쿼리에 대한 답변을 포함하는 **간결하고 관련성 높은 텍스트 스니펫**을 추출하여 제공합니다.

---

4. 최종 응답 제공 (Final Output Generation)

최종적으로, Tavily API는 랭킹된 상위 결과(링크와 스니펫)를 사용자에게 반환합니다.

- **간결한 출력:** 특히 LLM 통합을 위해 설계되었기 때문에, 불필요한 데이터를 제외하고 LLM이 **RAG(검색 증강 생성)** 과정에 바로 사용할 수 있도록 **최적화된 형식**으로 데이터를 전달합니다.
-

핵심 기능: `TavilySearchResults` 와 같은 LangChain 도구는 이 과정에서 랭킹된 상위 몇 개의 결과를 가져와 LLM이 질문에 답할 때 참조할 수 있도록 합니다.

요약하자면, Tavily Search는 여러 검색 엔진의 장점을 **AI 기반으로 취합**하고 **랭킹을 최적화**하여, LLM에게 빠르고 정확하며 검증된 정보를 공급하는 **중개자 역할**을 수행하는 것이 핵심입니다.

Tavily Search는 **검색 엔진이라기보다는 검색 데이터를 제공하는 API 또는 검색 에이전트**로 분류하는 것이 더 정확합니다.

- **Tavily는 자체적으로 웹 전체를 크롤링하여 거대한 독립 인덱스를 구축하지 않습니다.**
- 대신, 질의를 받으면 **다른 고품질 검색 엔진(Google, Bing 등)의 API**를 통해 실시간으로 데이터를 가져옵니다.
- 이후 AI를 사용하여 이 여러 소스의 결과를 통합하고, 질문에 가장 적합한 스니펫을 추출하여 개발자나 LLM에 최적화된 형태로 제공합니다.

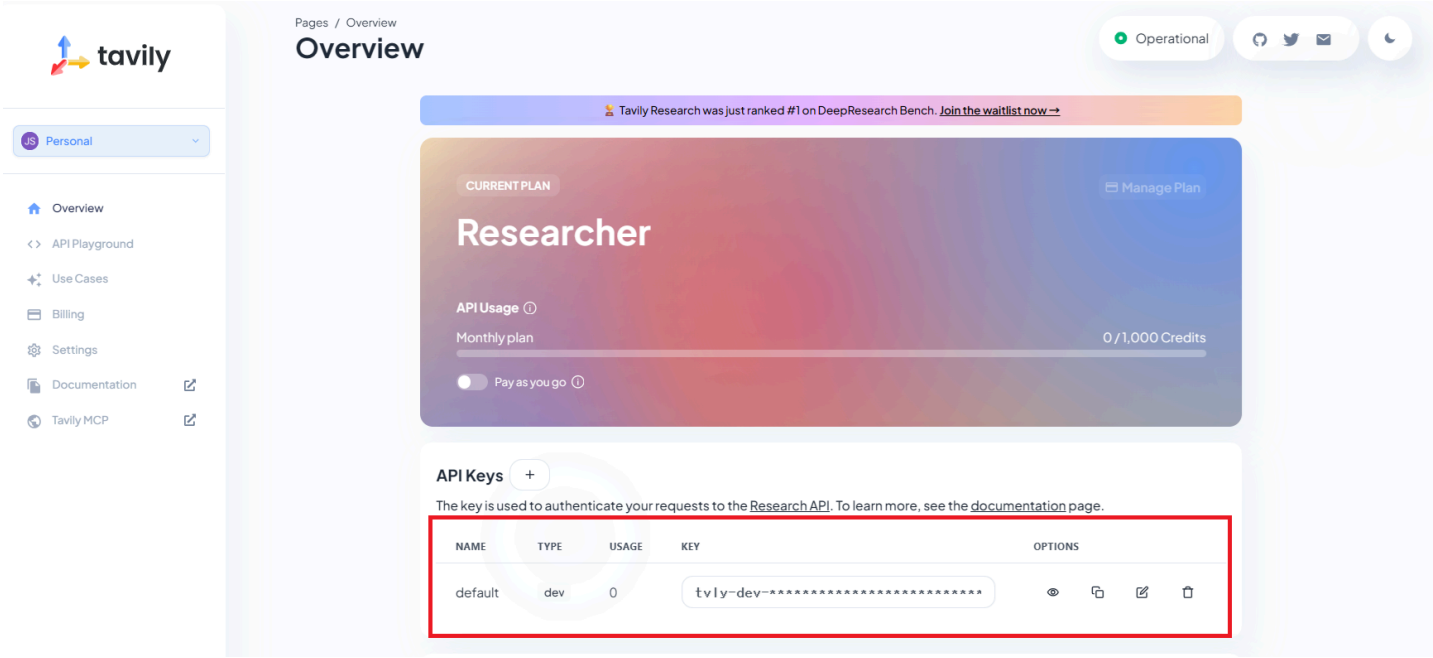
Tavily 사용하기

Tavily를 사용하기 위해서는 먼저 API 키를 발급 받아야 합니다. Tavily 웹사이트에서 회원 가입 후 API 키를 얻을 수 있습니다.

가입하기

https://www.tavily.com

가입 후 로그인 하면 바로 API Key를 얻을 수 있습니다.



TavilySearchResults

설명

- Tavily 검색 API를 쿼리하고 JSON 형식의 결과를 반환합니다.
- 포괄적이고 정확하며 신뢰할 수 있는 결과에 최적화된 검색 엔진입니다.

- 현재 이벤트에 대한 질문에 답변할 때 유용합니다.

주요 매개변수

- `max_results` (int): 반환할 최대 검색 결과 수 (기본값: 5)
- `search_depth` (str): 검색 깊이 ("basic" 또는 "advanced")
- `include_domains` (List[str]): 검색 결과에 포함할 도메인 목록
  ex) include_domains=["github.io", "wikidocs.net"],
- `exclude_domains` (List[str]): 검색 결과에서 제외할 도메인 목록
- `include_answer` (bool): 원본 쿼리에 대한 짧은 답변 포함 여부
- `include_raw_content` (bool): 각 사이트의 정제된 HTML 콘텐츠 포함 여부
- `include_images` (bool): 쿼리 관련 이미지 목록 포함 여부

Auto (Python)

```python
# 뉴스 검색 예제
result = tavily_tool.search(
    query="최신 AI 기술 동향",  # 검색 쿼리
    search_depth="basic",  # 기본 검색 수준
    topic="news",  # 뉴스 주제
    days=3,  # 최근 3일 내 결과
    max_results=5,  # 최대 5개 결과
    include_answer=False,  # 답변 미포함
    include_raw_content=False,  # 원본 콘텐츠 미포함
    include_images=False,  # 이미지 미포함
    format_output=True,  # 결과 포맷팅
)
```

반환 값

- 검색 결과를 포함하는 JSON 형식의 문자열(url, content)

설치

https://github.com/tavily-ai/tavily-python

Auto

```
pip install tavily-python langchain-community
```

실습 13_travily.py

Python

```python
from langchain_community.tools.tavily_search import TavilySearchResults

from dotenv import load_dotenv
load_dotenv()

tool = TavilySearchResults(max_results=1)
result = tool.invoke("antigravity는 무엇이고 최근 이슈는 무엇인가요?")

print(result)
```

출력

Auto (CSS)

```
[{'title': '구글 안티그래비티: 구글의 에이전트 우선 코딩 플랫폼 내부 - Macaron',
'url': 'https://macaron.im/ko/blog/google-antigravity-coding-agent',
'content': "Google Antigravity는 최근 출시된 AI 지원 소프트웨어 개발 플랫폼으로, 코드의 '에이전트 우선' 시대를 위한 것입니다. 간단히 말해, \\\\AI 에이전트로 강화된 IDE(통합 개발 환경)\\\\입니다. 단순히 코드 자동 완성에 그치지 않고, 이 AI 에이전트는  계획, 작성, 테스트, 심지어 여러 도구를 통해 코드를 실행할 수도 있습니다. Google은 Antigravity를 개발자가 “더 높은 수준의 작업 지향적 수준에서 작업할 수 있도록” 하는 플랫폼이라고 설명합니다. 즉, 원하는 목표를 AI에 알려주면 에이전트가 그것을 어떻게 달성할지 알아냅니다. 이 과정에서 여전히 IDE처럼 친숙 하게 느껴져 개발자는 필요할 때 전통적인 코드 작성 방식으로 작업할 수 있습니다. 목표는 AI를 수동적 보조가 아닌 능동적 코딩 파트너로 만드는 것입니다. [...] 이 프로젝트를 'Antigravity'라고 명명함으로써, Google은 대담하고 미래지향적인 혁신의 이미지를 의도적으로 불러일으킵니다. 이는 Google X '문샷 팩토리' 정신을 연상시킵니다 – 소행성 채굴, 우주 엘리베이터, 자율주행차와 같은 대담한 아이디어를 추구하는  곳입니다. Antigravity는 소프트웨어 도구이지만, 전통적 제약에서 벗어나려는 그 정신을 담고 있습니다.  전통적인 소프트웨어 엔지니어링에서는 더 많은 기능을 추가하거나 복잡한 시스템을 구축하는 것이 대개 유지해야 할 코드가 더 많아지고, 수정해야 할 버그가 더 많아지면서 중력에 눌리게 됩니다. Google Antigravity는 그 무게를 제거하려 하며, 개발자가 덜 억압받으면서 더 많이 구축할 수 있도록 합니다. 이것은 실험적인 아이디어입니다: 코딩에 중력이 없다면, 탈출 속도로 움직일 수 있다면 어떨까요? [...] 현재로서는  호기심 많은 개발자들이 Antigravity를 무료로 다운로드하여 체험해볼 수 있습니다. 당신이 잡무를 덜어내 고 싶은 전문 개발자이든 AI에 관심 있는 취미 개발자이든, 앱을 '런칭'하고 실험해보는 것은 가치가 있습 니다. 그 이름 자체가 탐험을 초대합니다: Antigravity는 일반적인 규칙이 완전히 적용되지 않는다는 것을 암시합니다. 실제로 AI 에이전트가 여러분을 대신해 코드를 작성하고 테스트하는 것을 보면서 마치 실시간 으로 중력이 무시되는 것을 보는 듯한 신비로운 느낌을 받을 수 있습니다. 이는 기술 발전을 지속시키는 혁신적이고 과학적으로 기반을 둔 놀이를 잘 보여줍니다. Google Antigravity는 우리 모두에게 흥미로운 질문을 던집니다: 소프트웨어 개발 자체가 거의 무중력 상태가 된다면 우리는 무엇을 만들게 될까요? \n\n## 참 고 자료 (출처)", 'score': 0.9930962}]
```

고급 기능: 검색 모드 설정

Tavily는 다양한 검색 모드를 제공합니다. 예를 들어, 뉴스 검색 모드를 사용하려면 다음과 같이 설정할 수 있습니다:

Auto (Bash)

```
retriever = TavilySearchAPIRetriever(
    k=3, search_depth="advanced", include_domains=["news"]
)
```

topic Available options:

`general` ,

`news` ,

`finance`

search_depth Available options:

`basic` ,

`advanced`

https://docs.tavily.com/documentation/api-reference/endpoint/search

이름	유형	작동 방식	주요 목적
Brave Search	**독립적 검색 엔진**	자체 웹 크롤링 및 인덱스 구축	일반 사용자의 웹 검색 (개인 정보 보호 중시)
Google Search	**독립적 검색 엔진**	자체 웹 크롤링 및 인덱스 구축	일반 사용자의 웹 검색 (가장 큰 인덱스)
Bing Search	**독립적 검색 엔진**	자체 웹 크롤링 및 인덱스 구축	일반 사용자의 웹 검색 (Microsoft의 서비스)
Tavily Search	**검색 API/에이전트**	다른 검색 엔진의 결과 통합 및 AI 최적화	LLM 및 애플리케이션 개발자의 RAG 시스템 통합

코드 인터프리터 도구 활용

Python REPL 도구

Python REPL 도구 설명: `PythonREPLTool`

`PythonREPLTool` 은 LangChain에서 제공하는 도구(Tool) 중 하나로, **에이전트나 애플리케이션이 파이썬(Python) 코드를 직접 실행하고 그 결과를 활용할 수 있도록** 설계되었습니다. 이는 LLM(대규모 언어 모델)이 복잡한 계산을 수행하거나, 데이터 조작을 하거나, 논리적인 문제 해결을 위해 실제 코드를 작성하고 테스트할 수 있는 환경을 제공합니다.

---

1. `PythonREPLTool` 의 주요 역할

이 도구는 Python의 **REPL (Read-Eval-Print Loop)** 환경을 래핑(Wrapping)하여 제공합니다.

- **Read (읽기):** 사용자의 입력(Python 명령어 문자열)을 받습니다.
- **Eval (평가/실행):** 그 명령어를 실제 Python 셸 환경에서 실행합니다.
- **Print (출력):** 실행 결과를 문자열로 반환합니다.

💡 사용 시 주의사항

결과를 명시적으로 출력해야 합니다. 도구 실행 결과로 값을 받으려면 입력 코드 내에서 `print( ... )` 함수를 명시적으로 사용해야 합니다. 예를 들어, `2 + 2` 만 입력하면 아무 결과도 반환되지 않으며, `print(2 + 2)` 라고 입력해야 결과인 `4` 가 반환됩니다.

---

2. 주요 특징 및 매개변수

특징/매개변수	설명	기본값
`sanitize_input`	입력으로 들어온 코드 문자열을 실행하기 전에 **정제**할지 여부를 결정합니다.	`True`
`python_repl`	도구 내부에서 실제로 코드를 실행하는 `PythonREPL` 인스턴스입니다.	전역 범위에서 실행되는 기본 인스턴스

입력 정제 (`sanitize_input`)

`sanitize_input` 이 `True` 인 경우, 도구는 사용자 입력 코드에서 불필요한 형식 요소를 제거합니다. 이는 특히 LLM이 코드를 생성할 때 자주 발생하는 Markdown 형식의 불필요한 문자를 처리하는 데 유용합니다.

- 제거 대상: 코드 블록을 감싸는 백틱(```), 'python' 또는 'py'와 같은 언어 지정자, 불필요한 공백 등이 제거됩니다.

3. 사용 방법

`PythonREPLTool` 은 일반적인 LangChain 도구와 동일하게 사용됩니다.

1. **인스턴스 생성:** `PythonREPLTool()` 을 호출하여 도구 인스턴스를 생성합니다.
2. **코드 실행:** 생성된 인스턴스의 `run` , `arun` , 또는 `invoke` 메서드에 실행하고자 하는 유효한 Python 코드 문자열을 입력으로 전달합니다.

Auto

```
pip install langchain-experimental
```

Auto (Haskell)

```
from langchain_experimental.tools import PythonREPLTool

# 1. 도구 인스턴스 생성
python_tool = PythonREPLTool()

# 2. 코드 실행 (print() 사용 필수)
code_to_run = "import math; print(math.sqrt(144) + 5)"
result = python_tool.run(code_to_run)

# result에는 '17.0\n'과 같은 실행 결과가 포함됩니다.
```

이 도구는 LLM에게 단순한 텍스트 처리 능력을 넘어 **실제 컴퓨팅 능력**을 부여하여 에이전트의 문제 해결 범위를 획기적으로 확장시킵니다.

사용자 정의 도구(Custom Tool)

LangChain 에서 제공하는 빌트인 도구 외에도 사용자가 직접 도구를 정의하여 사용할 수 있습니다.

@tool 데코레이터

이 데코레이터는 함수를 도구로 변환하는 기능을 제공합니다. 다양한 옵션을 통해 도구의 동작을 커스터마이즈할 수 있습니다.

사용 방법

1. 함수 위에 `@tool` 데코레이터 적용
2. 필요에 따라 데코레이터 매개변수 설정

이 데코레이터를 사용하면 일반 Python 함수를 강력한 도구로 쉽게 변환할 수 있으며, 자동화된 문서화와 유연한 인터페이스 생성이 가능합니다.

LangChain의 `@tool` 데코레이터의 매개변수 설정이란, 단순히 함수 위에 `@tool` 을 붙이는 것을 넘어, **도구(Tool)로 변환되는 함수의 속성을 사용자가 직접 지정하여 대규모 언어 모델(LLM)이 해당 도구를 더 정확하고 유연하게 사용하도록 돕는 것**을 의미합니다.

기본적으로 `@tool` 데코레이터는 다음 정보를 함수로부터 자동으로 추출합니다.

- **도구 이름 (name):** 함수의 이름을 사용합니다.
  ex) @tool("웹_검색")
- **도구 설명 (description):** 함수의 **docstring** (문자열)을 사용합니다. 이 설명은 LLM이 도구를 언제 사용해야 하는지 판단하는 데 결정적인 역할을 합니다.
  ex) @tool(description="현재 날씨 정보를 조회합니다.")
- **입력 스키마 (args_schema):** 함수의 **매개변수 주석(Type Hinting)**을 사용하여 도구의 입력 형식을 자동으로 생성합니다.

- @tool(return_direct=True)

  도구의 출력을 **즉시 사용자에게 반환**하고 에이전트의 생각을 멈추게 할 때 사용합니다. 예를 들어, 계산기 도구의 결과는 보통 추가적인 사고 없이 바로 사용자에게 전달하는 것이 좋습니다.

`return_direct=True` 작동 방식 및 예시

1. `return_direct=False` (기본값)의 동작 (비효율적)

대부분의 도구는 이 기본값으로 설정됩니다. 도구 사용 후, 에이전트는 결과를 받아 **다음 행동을 결정**해야 합니다.

단계	동작	에이전트의 '생각 (Thought)'
사용자 요청	"서울의 현재 온도는?"	
Action	`weather_tool(city="서울")` 실행	
Observation	`서울의 현재 온도는 15°C 입니다.`	

Thought	"도구로부터 날씨 정보를 얻었다. 이제 이 정보를 최종 답변 형식으로 사용자에게 제공해야 한다."	**불필요한 사고**
Final Answer	"서울의 현재 온도는 15°C 입니다."	

이 경우, `15°C` 라는 단순한 결과가 나왔음에도 에이전트는 추가적인 LLM 호출을 통해 사고하고 답변을 구성하는 **추가적인 시간과 비용**을 소모합니다.

---

2. `return_direct=True` 의 동작 (효율적)

단순하고 최종적인 결과를 내는 도구(예: 계산기, 단순 조회 도구)에 이 옵션을 설정합니다.

Auto (Python)

```python
from langchain.tools import tool

@tool(return_direct=True) # 이 옵션이 핵심입니다.
def calculator(expression: str) → float:
    """수학적 표현식을 계산합니다. (예: 5 + 3 * 2)"""
    return eval(expression) # 예시를 위한 단순화

# ... 에이전트 실행 ...
```

단계	동작	에이전트의 '생각 (Thought)'
사용자 요청	"123 + 456을 계산해줘."	
Action	`calculator(expression="123 + 456")` 실행	
Observation	`579`	
최종 응답 즉시 반환	**"579"**	**👉 추가적인 사고 없이 즉시 종료 및 반환**

`return_direct=True` 를 사용하면, 도구 실행 결과 `579` 가 나오자마자 에이전트 루프가 종료되고 해당 값이 바로 사용자에게 전달됩니다.

요약:

이 옵션은 **단순 조회/계산 도구**에서 불필요한 LLM 호출을 제거하여 **응답 속도를 높이고 운영 비용을 절감**하는 데 결정적인 역할을 합니다.

13_custom_tool.py

Auto (Python)

```python
from langchain.tools import tool


# 데코레이터를 사용하여 함수를 도구로 변환합니다.
@tool
def add_numbers(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b


@tool("곱셈연산")
def multiply_numbers(a: int, b: int) -> int:
    """Multiply two numbers"""
    return a * b

# 도구 실행
add_result = add_numbers.invoke({"a": 3, "b": 4})

# 도구 실행
multiply_result = multiply_numbers.invoke({"a": 3, "b": 4})

print(f"도구 이름: {add_numbers.name}")   # add_numbers
print(f"도구 설명: {add_numbers.description}") # Add two numbers
print(f"도구 결과: {add_result}")

# 사용자가 입력한 값을 도구에 전달
print(f"도구 이름: {multiply_numbers.name}")   # multiply_numbers
print(f"도구 결과: {multiply_result}")

# LangChain이 생성한 args_schema를 확인해봅니다.
schema = add_numbers.args # .args 속성을 통해 스키마에 접근 가능
print(schema)
```

LLM 도구 바인딩(Binding Tools)

LLM 모델이 도구(tool) 를 호출할 수 있으려면 chat 요청을 할 때 모델에 도구 스키마(tool schema) 를 전달해야 합니다.

도구 호출(tool calling) 기능을 지원하는 LangChain Chat Model 은 `.bind_tools()` 메서드를 구현하여 LangChain 도구 객체, Pydantic 클래스 또는 JSON 스키마 목록을 수신하고 공급자별 예상 형식으로 채팅 모델에 바인딩(binding) 합니다.

- Pydantic 라이브러리는 주로 **데이터 모델을 정의하는데 사용**

llm 모델에 `bind_tools()` 를 사용하여 도구를 바인딩합니다.

결과는 `tool_calls` 에 저장됩니다. 따라서, `.tool_calls` 를 확인하여 도구 호출 결과를 확인할 수 있습니다.

Auto (Python)

```python
from langchain_ollama import ChatOllama

tools = [add_numbers, multiply_numbers, python_tool]

llm = ChatOllama(
    base_url="http://localhost:11434",  # Ollama 서버 주소
    model="gemma3:1b",
    temperature=0.3,
)

llm_with_tools = llm.bind_tools(tools)
llm_with_tools.invoke("What is the length of the word 'teddynote'?").tool_calls
```

`create_tool_calling_agent` 설명

`create_tool_calling_agent` 함수는 LangChain Expression Language (LCEL)에서 사용되는 핵심 함수로, LLM 모델을 활용하여 **도구 호출(Tool Calling) 체인**을 쉽게 구성할 수 있도록 돕습니다.

이 함수를 사용하면 LLM이 사용자의 요청을 이해하고, 요청을 처리하는 데 필요한 도구(Tool)를 식별하여 올바른 형식으로 호출할 수 있는 에이전트 역할을 수행하게 됩니다.

---

1. `create_tool_calling_agent` 의 역할

이 함수는 LLM, 도구 목록, 프롬프트 템플릿을 결합하여, **도구 사용을 지시하는 특정 구조를 가진 에이전트 프롬프트**를 만듭니다.

- **입력:** 사용자의 질문 (User Input)
- **출력:** 다음 단계로 전달할 **에이전트 상태(Agent State)** 또는 **LLM의 응답**

핵심 기능은 LLM이 사용자 질문을 분석하여, **어떤 도구를 어떤 인자(arguments)와 함께 호출해야 하는지**에 대한 JSON 형식의 응답을 생성하도록 유도하는 것입니다.

2. 주요 구성 요소

`create_tool_calling_agent` 를 호출할 때 필요한 세 가지 핵심 구성 요소는 다음과 같습니다.

구성 요소	설명	역할
`llm`	**도구 호출을 지원하는 LLM**	사용자 질문을 분석하고, 사용할 도구와 인수를 JSON 형식으

		로 결정합니다. (예: `ChatOpenAI(model="gpt-4o")`)
`tools`	**사용 가능한 도구 목록**	LLM이 접근할 수 있는 모든 `BaseTool` 객체의 리스트입니다. (예: `TavilySearchResults`, `GoogleSearch`, 커스텀 도구)
`prompt`	**Agent 프롬프트 템플릿**	LLM에게 전반적인 임무, 시스템 지침, **도구를 사용하는 방법**을 알려주는 `ChatPromptTemplate`입니다. 이 템플릿에는 반드시 `tools`, `tool_names`, `agent_scratchpad` 변수가 포함되어야 합니다.

3. 작동 흐름 (Agent Executor 내부)

`create_tool_calling_agent`로 생성된 로직은 보통 `AgentExecutor` 내에서 실행되며, 그 과정은 다음과 같습니다.

1. **사용자 질문 입력:** 사용자가 질문을 던집니다.
2. **LLM 호출:** `prompt`, `tools`, 그리고 이전 실행 결과(`agent_scratchpad`)와 함께 LLM을 호출합니다.
3. **LLM 결정:** LLM은 두 가지 중 하나를 결정합니다.
   - **Final Answer (최종 답변):** 도구 없이 바로 답변할 수 있다면 최종 답변을 출력합니다.
   - **Tool Call (도구 호출):** 정보가 필요하다고 판단하면, **도구 이름**과 **호출 인자**를 JSON 형식으로 포함하는 응답을 반환합니다.
4. **도구 실행 (Action):** `AgentExecutor`는 LLM이 요청한 도구를 실행하고 결과를 받습니다.
5. **결과 반영:** 도구 실행 결과는 `agent_scratchpad`에 저장되어 (Observation) 다음 LLM 호출의 입력으로 다시 들어갑니다. (반복)
6. **반복 종료:** LLM이 최종 답변을 출력할 때까지 3~5단계를 반복합니다.

`agent_scratchpad`는 LangChain 에이전트 시스템에서 에이전트가 **현재 목표를 달성하기 위해 수행한 중간 작업과 그 결과들을 임시로 기록하는 공간** 또는 **임시 작업 메모리** 역할을 합니다.

이는 에이전트의 반복적인 사고 및 행동 루프를 가능하게 하는 핵심적인 구성 요소입니다.

`agent_scratchpad`의 역할과 필요성

`agent_scratchpad`는 LLM이 **"이전에는 무슨 일을 했고, 그 결과는 무엇이었는지"**를 파악하여 다음 행동을 결정할 수 있도록 도와주는 일종의 **로그(Log)** 또는 **작업 기록부**입니다.

1. 작동 원리 (기억의 주입)

- **저장되는 내용:** Agent Executor가 도구를 호출(Action)하고 그 결과를 받아오면(Observation), 이 **'Action과 Observation 쌍'**이 순차적으로 `agent_scratchpad`에 기록됩니다.
- **LLM으로 전달:** Agent Executor가 LLM을 다시 호출할 때, 이 `agent_scratchpad`는 **프롬프트의 일부**로 포함되어 LLM에게 전달됩니다.
- **사고 과정 재개:** LLM은 스크래치패드에 담긴 이전의 모든 시도와 결과를 검토하고, 이를 바탕으로 "지금까지 이랬으니, 다음에는 어떤 도구를 어떻게 호출해야 최종 답변을 얻을 수 있겠다"는 **새로운 사고(Thought)**를 시작하게 됩니다.

2. 필요한 이유 (연속적인 추론)

에이전트 시스템의 가장 중요한 기능은 **여러 단계를 거쳐 복잡한 문제를 해결**하는 능력입니다.

- **예시:** "어제 날씨를 조회하고, 그 온도를 화씨로 변환해줘."
  - **1단계:** 에이전트는 `날씨_조회_도구` 를 사용합니다. → 결과가 스크래치패드에 기록됩니다.
  - **2단계:** 에이전트는 스크래치패드의 조회 결과를 보고, 이제 `온도_변환_도구` 를 사용해야 한다고 판단합니다.
  - **3단계:** 변환된 최종 결과를 얻습니다.

`agent_scratchpad` 가 없다면, LLM은 매 호출마다 "어제 날씨"를 다시 조회하는 등 **이전 단계를 기억하지 못해 비효율적으로 반복**하거나, 아예 복합적인 작업을 수행할 수 없게 됩니다.

4. 사용 예시 (LCEL 파이프라인)

Auto (Python)

```python
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain_community.tools.tavily_search import TavilySearchResults

# 1. 도구 정의
tools = [TavilySearchResults(max_results=3)]

# 2. LLM 정의 (OpenAI의 도구 호출 기능 사용)
llm = ChatOpenAI(model="gpt-4o", temperature=0)

# 3. 프롬프트 정의 (Hub에서 기본 템플릿 로드)
prompt = hub.pull("hwchase17/openai-tools-agent")

# 4. Agent 생성
agent = create_tool_calling_agent(llm, tools, prompt)

# 5. Agent Executor 생성 (실행 환경)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# 실행
# result = agent_executor.invoke({"input": "최근 며칠간 달 탐사선 이슈에 대해 알려줘"})
```

참고

<랭체인LangChain 노트> - LangChain 한국어 튜토리얼KR : https://wikidocs.net/book/14314