
- 출처: LangChain 공식 문서 또는 해당 교재명
- 원본 URL: https://smith.langchain.com/hub/teddynote/summary-stuff-documents

---

3. `도구 호출 에이전트` *(Tool Calling Agent)*

1) 개념

- 도구 호출 사용의 이점
  - 모델이 하나 이상의 `도구` **(tool)** 가 **호출되어야 하는 시기를 감지하고 해당 도구에 전달해야 하는 입력** 으로 전달 가능
  - API 호출에서 도구를 설명하고 모델이 이러한 도구를 호출하기 위한 인수가 포함된 JSON과 같은 구조화된 객체를 출력하도록 지능적으로 선택 가능
- 도구 API의 목표: 일반 텍스트 완성이나 채팅 API를 사용하여 수행할 수 있는 것보다 **더 안정적으로 유효하고 유용한** `도구 호출` **(tool call)** 을 반환하는 것
- 구조화된 출력을 도구 호출 채팅 모델에 여러 도구를 바인딩하고 모델이 호출할 도구를 선택할 수 있다는 사실과 결합하여 쿼리가 해결될 때까지 반복적으로 도구를 호출하고 결과를 수신하는 에이전트를 만들 수 있음
  - 
  - OpenAI의 특정 도구 호출 스타일에 맞게 설계된 OpenAI 도구 에이전트 보다 `일반화된 버전`
  - **에이전트** `LangChain` **의** `ToolCall` **인터페이스를 사용 →** `OpenAI`, `Anthropic`, `Google Gemini`, `Mistral` 과 같은 더 광범위한 공급자 구현 지원
  - `참고 링크`

- `환경설정`

```
# API 키를 환경변수로 관리하기 위한 설정 파일
from dotenv import load_dotenv

# API 키 정보 로드
load_dotenv()                                   # True
```

```
from langsmith import Client
from langsmith import traceable
```

```
import os

# LangSmith 환경 변수 확인

print("\n--- LangSmith 환경 변수 확인 ---")
langchain_tracing_v2 = os.getenv('LANGCHAIN_TRACING_V2')
langchain_project = os.getenv('LANGCHAIN_PROJECT')
langchain_api_key_status = "설정됨" if os.getenv('LANGCHAIN_API_KEY') else "설정되

if langchain_tracing_v2 == "true" and os.getenv('LANGCHAIN_API_KEY') and langc
    print(f"✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='{langchain_tracing_v
    print(f"✅ LangSmith 프로젝트: '{langchain_project}'")
    print(f"✅ LangSmith API Key: {langchain_api_key_status}")
    print("  -> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.")
else:
    print("❌ LangSmith 추적이 완전히 활성화되지 않았습니다. 다음을 확인하세요:")
    if langchain_tracing_v2 != "true":
        print(f"  - LANGCHAIN_TRACING_V2가 'true'로 설정되어 있지 않습니다 (현재: '{la
    if not os.getenv('LANGCHAIN_API_KEY'):
        print("  - LANGCHAIN_API_KEY가 설정되어 있지 않습니다.")
    if not langchain_project:
        print("  - LANGCHAIN_PROJECT가 설정되어 있지 않습니다.")
```

- 셀 출력

```
--- LangSmith 환경 변수 확인 ---
✅ LangSmith 추적 활성화됨 (LANGCHAIN_TRACING_V2='true')
✅ LangSmith 프로젝트: 'LangChain-prantice'
✅ LangSmith API Key: 설정됨
-> 이제 LangSmith 대시보드에서 이 프로젝트를 확인해 보세요.
```

```
# ========================================
# 경고 메시지 무시
# ========================================
import os
os.environ['TOKENIZERS_PARALLELISM'] = 'false'
```

```
import sys
from pathlib import Path

# 루트 디렉토리를 Python 경로에 추가
root_dir = Path().absolute().parent
sys.path.append(str(root_dir))

print(f"✅ 루트 디렉토리 추가: {root_dir}")

# config import
from config import get_llm, get_embeddings
from config import gpt_5_nano, gpt_5_mini

print("✅ config.py import 성공!")
```

- 응답 시간: `5.0s`
- ✅ 루트 디렉토리 추가: `루트/20250727-langchain-note`
- ✅ config.py import 성공!

```
from langchain_google_genai import ChatGoogleGenerativeAI

# API 키 확인
if not os.getenv("GOOGLE_API_KEY2"):
    os.environ["GOOGLE_API_KEY2"] = input("Enter your Google API key: ")

# LLM 초기화
gemini_lc = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,                                          # temperature
    max_output_tokens=4096,
    )

result=gemini_lc.invoke("대한민국의 수도는?")
print(result.content)
```

- 셀 출력: `1.9s`

```
대한민국의 수도는 **서울**입니다.
```

- `2.0s`
- `GoogleNews` 라이브러리 직접 사용 위해 설치

```
%pip install GoogleNews
```

- `4.1s`
- `langchain-experimental` 패키지 설치

```
%pip install langchain-experimental
```

```
from langchain.tools import tool
from typing import List, Dict, Annotated
from GoogleNews import GoogleNews
from langchain_experimental.utilities import PythonREPL


# 도구 생성
```

```
@tool
def search_news(query: str) -> List[Dict[str, str]]:
    """Search Google News by input keyword"""
    news_tool = GoogleNews()

    # 1. 검색 실행
    # 메서드 이름 변경하기: search_by_keyword -> search
    news_tool.search(query)


    # 2. 결과 반환 방식 변경하기: (search_by_keyword() -> result())
    # ERROR: return news_tool.search_by_keyword()[:5]
    return news_tool.result()[:5]
```

```
# 도구 생성
@tool
def python_repl_tool(
    code:Annotated[str, "The python code to execute to generate your chart."],
):
    """Use this to execute python code. If you want to see the output of a val
    result = ""
    try:
        result = PythonREPL().run(code)
    except BaseException as e:
        print(f"Failed to execute. Error: {repr(e)}")
    finally:
        return result

print(f"① 도구 이름: {search_news.name}")
print(f"① 도구 설명: {search_news.description}")
print(f"② 도구 이름: {python_repl_tool.name}")
print(f"② 도구 이름: {python_repl_tool.description}")
```

- 셀 출력

```
① 도구 이름: search_news
① 도구 설명: Search Google News by input keyword
② 도구 이름: python_repl_tool
② 도구 이름: Use this to execute python code. If you want to see the output of
```

```
# tool 정의

tools = [search_news, python_repl_tool]
```

2) `Agent` 프롬프트 생성

- **`chat_history`**: 이전 대화 내용을 저장하는 변수
  - *멀티턴을 지원하지 않는다면, 생략 가능*
- **`agent_scratchpad`**: 에이전트가 임시로 저장하는 변수

- `input`: 사용자의 입력

```
from langchain_core.prompts import ChatPromptTemplate

# 프롬프트 생성
# 프롬프트 = 에이전트에게 모델이 수행할 작업을 설명하는 텍스트를 제공함
# 도구의 이름과 역할을 입력해야 함

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are a helpful assistant. "
            "Make sure to use the `search_news` tool for searching keyword rel
        ),
        ("placeholder", "{chat_history}"),
        ("human", "{input}"),
        ("placeholder", "{agent_scratchpad}"),
    ]
)                                                    # 0.1s
```

3) Agent 생성

- `langchain이 v1.0.5`로 업데이트되면서 **`Agent` 생성 방법이 `바뀜`**
  - 임포트 방법 바뀜
    - `from langchain.agents import create_tool_calling_agent` → **`from langchain.agents import create_agent`**
  - **`prompt`** 변수 = **문자열로만 전달해야 함** → 따라서 **`프롬프트의 내용 중 시스템 지시사항만을 문자열로 전달`** 하는 것이 가장 깔끔하고 올바른 방법
    - 위의 `prompt` 변수를 `PromptTemplate` 그대로 사용해서는 안됨
    - **`타입 불일치`**: **`str`** **(문자열)** 기대하기 때문 ↔ 반면에 **`PromptTemplate`** = 복잡한 객체
    - **`기능 중복`**
      - 위 템플릿에 있는 `{agent_scratchpad}`나 `{chat_history}` 같은 플레이스홀더들은 **`새로운 create_agent 함수`가 내부적으로 알아서 관리**
      - 전체 템플릿을 넘기면 이 기능들이 충돌하거나 의도대로 동작하지 않을 수 있음

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import create_agent

# LLM 정의
llm = gemini_lc

# 시스템 메시지 내용만 문자열로 정의
system_instruction = (
    "You are a helpful assistant. "
```

```
    "Make sure to use the `search_news` tool for searching keyword related new
)

# Agent 생성 (v1.0.5 기준)
agent = create_agent(
    model=llm,
    tools=tools,
    #system_prompt=system_instruction,
    )                           # 0.1s
```

4) AgentExecutor

- `AgentExecutor` = 도구를 사용하는 에이전트를 실행하는 클래스
- `v.1.0.5` 이후 버전부터는 레거시 구성요소로 분류되어 **`langchain` 기본 패키지에서 제외됨 → `langchain_classic` 패키지로 이동함**
    - 해당 기능을 사용하기 위해서는 **`import 경로를 수정해야 함`**

- **`주요 속성`**
    - **`agent`**: 실행 루프의 각 단계에서 계획을 생성하고 행동을 결정하는 에이전트
    - **`tools`**: 에이전트가 사용할 수 있는 유효한 도구 목록
    - **`return_intermediate_steps`**: 최종 출력과 함께 에이전트의 중간 단계 경로를 반환할지 여부
    - **`max_iterations`**: 실행 루프를 종료하기 전 최대 단계 수
    - **`max_execution_time`**: 실행 루프에 소요될 수 있는 최대 시간
    - **`early_stopping_method`**: 에이전트가 AgentFinish를 반환하지 않을 때 사용할 조기 종료 방법. ("force" or "generate")
        - **`"force"`** = 시간 or 반복 제한에 도달하여 중지되었다는 문자열 반환
        - **`"generate"`** = 에이전트의 LLM 체인을 마지막으로 한 번 호출하여 이전 단계에 따라 최종 답변을 생성
    - **`handle_parsing_errors`**
        - 에이전트의 출력 파서에서 발생한 오류 처리 방법
        - `True`, `False`, 또는 `오류 처리 함수`
    - **`trim_intermediate_steps`**
        - 중간 단계를 트리밍하는 방법
        - `-1 trim 하지 않음` or `트리밍 함수`

- **`주요 메서드`**
    - **`invoke`**: 에이전트 실행

    - `stream`: 최종 출력에 도달하는 데 필요하는 단계를 *스트리밍*

- `주요 기능`
    - `도구 검증`: 에이전트와 호환되는 도구인지 확인
    - `실행 제어`: 최대 반복 횟수 및 실행 시간 제한 설정 가능
    - `오류 처리`: 출력 파싱 오류에 대한 다양한 처리 옵션 제공
    - `중간 단계 관리`: 중간 단계 트리밍 및 반환 옵션
    - `비동기 지원`: 비동기 실행 및 스트리밍 지원

- `최적화 팁`
    - `max_iterations` 와 `max_execution_time` 을 적절히 설정하여 실행 시간 관리
    - `trim_intermediate_steps` 를 활용하여 메모리 사용량 최적화
    - 복잡한 작업의 경우 `stream` 메서드를 사용하여 단계별 결과 모니터링

- 기존 교재 안내된 코드로 실행하기 → *`ERROR`* 발생

```
# 기존: from langchain.agents import AgentExecutor
# v1.0.5 버전으로 임포트하기
from langchain_classic.agents import AgentExecutor
from langchain_core.messages import HumanMessage

# AgentExecutor 생성하기
agent_executor = AgentExecutor(
    agent = agent,
    tools = tools,
    verbose = True,
    max_iterations = 10,
    #max_execution_time = 10,
    recursion_limit = 10,
    handle_parsing_errors = True,
)                                                    # 3.8s


# AgentExecutor 실행
result = agent_executor.invoke({"input":"AI 투자와 관련된 뉴스를 검색해주세요."})

print("Agent 실행 결과: ")
print(result["output"])
```

- `v.1.0.5` 버전의 `agent` 객체는 이미 스스로 실행 계획을 세우고 도구를 호출하는 모든 기능을 갖고 있음
    - 임포트 수정하기 → `result` 수정해서 실행하기

```
from langchain_core.messages import HumanMessage

# 실행 시 입력 형식이 "messages" 리스트여야 함
result = agent.invoke(
    {"messages": [HumanMessage(content="AI 투자와 관련된 뉴스를 검색해주세요.")]},
    # max_iterations 대신 recursion_limit 사용
    config={"recursion_limit": 10}
)

# 결과 출력
# result["messages"]에는 대화의 모든 중간 과정(생각, 도구 호출 등)이 포함되어 있음
print("=== 실행 결과 ===")
print(result["messages"][-1].content)
```

- 셀 출력 (`6.1s`)

```
=== 실행 결과 ===
AI 투자와 관련하여 다음과 같은 뉴스들을 찾아왔습니다:

*   "AI Hype Cools, Interest Rates Fall: How the Financial Times Sees the Glo
*   "2026년 고확신 투자 아이디어 – AI의 핵심 에너지 확보: 성장 잠재력 측면에서는 Eaton, 방어 측
*   "TSMC: AI로 인한 경기 침체에도 흔들리지 않는, 탄탄한 장기 투자 종목" – TSMC가 AI 관련 투자
*   "Ubitus Receives Major METI Investment Grant, Investing JPY 17 Billion to
*   "김민 카지노 배우 에서 성공하는 방법: 전문가들의 조언 – 성공을 위한 필수 요소" – 이 뉴스는 AI

이 중에서 더 자세히 알고 싶은 뉴스가 있으신가요?
```

5) Stream 출력으로 단계별 결과 확인

- AgentExecutor의 stream() 메소드를 사용하여 에이전트의 중간 단계를 스트리밍할 것
- `stream()` 의 출력은 (Action, Observation) 쌍 사이에서 번갈아 나타나며, 최종적으로 에이전트가 목표를 달성했다면 답변으로 마무리됨
    - ① `Action` 출력
    - ② `Observation` 출력
    - ③ `Action` 출력
    - ④ `Observation` 출력
    - ... (목표 달성까지 계속) ...

- 최종 목표가 달성되면 에이전트는 최종 답변을 출력할 것
- ⑤ `Answer` 출력

- 출력 내용 요약

출력	내용
Action	* actions: AgentAction 또는 그 하위 클래스 * messages: 액션 호출에 해당하는 채팅 메시지
Observation	* steps: 현재 액션과 그 관찰을 포함한 에이전트가 지금까지 수행한 작업의 기록 * messages: 함수 호출 결과(즉, 관찰)를 포함한 채팅 메시지
Final Answer	* output: AgentFinish * messages: 최종 출력을 포함한 채팅 메시지

```
from langchain_core.messages import HumanMessage

# Agent 바로 실행하기
# 스트리밍 모드
result = agent.stream(
    {"messages": [HumanMessage(content="AI 투자와 관련된 뉴스를 검색해주세요.")]},
    # max_iterations 대신 recursion_limit 사용
    config={"recursion_limit": 10}
)

for step in result:
    # 중간 단계 출력
    print(step)
```

- 셀 출력 (`2.9s`)

```
{'model': {'messages': [AIMessage(content='', additional_kwargs={'function_ca
{'tools': {'messages': [ToolMessage(content="[{'title': 'AI Hype Cools, Inter
{'model': {'messages': [AIMessage(content='AI 투자와 관련된 뉴스를 검색했습니다. 2026
```

6) 중간 단계 출력을 사용자 정의 함수로 출력

- 다음 3개의 함수를 정의 → 중간 단계 출력을 사용자 정의하기
  - **`tool_callback`**: 도구 호출 출력을 처리하는 함수
  - **`observation_callback`**: 관찰 (`Observation`) 출력을 처리하는 함수
  - **`result_callback`**: 최종 답변 출력을 처리하는 함수

- 사용자 정의 파서 설계하기
  - `LangGraph`가 반환하는 스트리밍 `chunk` = `dict`
    - 예시:

```
({'model': ...}, {'tools': ...})
```

- 실제 필요한 것 = `AIMessage`의 `content` 뿐
  - ① `AI의 답변` = (model 단계) 그 내용을 출력하기
  - ② `도구 실행 결과` = (tools 단계) → 필요하다면 출력 or 생략하기
    - 보통 최종 답변만 보려면 생략
    - 혹은 `"검색 중..."` 같은 메시지만 띄우기

```
from langchain_core.messages import AIMessage, ToolMessage

def parse_agent_stream(stream):
    """
    LangGraph Agent의 스트림 출력을 파싱하여 깔끔하게 보여주는 함수
    """
    for chunk in stream:
        # 1. 모델(LLM)의 응답 단계인 경우
        if "model" in chunk:
            model_messages = chunk["model"]["messages"]
            for message in model_messages:
                if isinstance(message, AIMessage):
                    # 도구 호출이 포함된 경우 (아직 최종 답변 아님)
                    if message.tool_calls:
                        tool_name = message.tool_calls[0]['name']
                        tool_args = message.tool_calls[0]['args']
                        print(f"🛠️ 도구 호출: {tool_name} (인자: {tool_args})")
                    # 최종 답변인 경우 (content가 있는 경우)
                    elif message.content:
                        print(f"🤖 AI 답변: {message.content}")

        # 2. 도구(Tool) 실행 결과 단계인 경우
        elif "tools" in chunk:
            tool_messages = chunk["tools"]["messages"]
            for message in tool_messages:
                if isinstance(message, ToolMessage):
                    print(f"✅ 도구 실행 완료 (결과 길이: {len(message.content)}자)"
```

- 스트리밍 방식으로 Agent 의 응답 과정을 확인하기

```
# 질의에 대한 답변을 스트리밍으로 출력 요청
result = agent.stream(
    {"messages" : [HumanMessage(content="matplotlib 을 사용하여 pie 차트를 그리는 코드
    config={"recursion_limit": 10}
)

# 사용자 정의 파서 함수로 출력하기
# 함수에 스트림 객체를 통째로 전달
# (함수 안에서 for 루프를 돌면서 하나씩 꺼내 처리함)
parse_agent_stream(result)
```

- 셀 출력 (`2.4s`)

```
WARNING:langchain_experimental.utilities.python:Python REPL can execute arbit
🛠️ 도구 호출: python_repl_tool (인자: {'code': "\nimport matplotlib.pyplot as pl
```

```
✅ 도구 실행 완료 (결과 길이: 51자)
```

- 실제 그래프가 그려지지 않는 문제가 발생
- 원인 분석
  - PythonREPL 도구는 코드를 실행하고 표준 출력(stdout) 결과를 문자열로 반환
  - 텍스트(print 결과 등)는 잘 나오지만, matplotlib으로 그린 그래프는 이미지 데이터이므로 터미널이나 일반적인 텍스트 출력으로는 볼 수 없음
  - PythonREPL 도구는 별도의 격리된(혹은 별도의) 파이썬 프로세스나 네임스페이스에서 실행되기 때문에 plt.show()를 해도 노트북 화면에 나타나지 않음
- 해결 방법: 에이전트가 생성한 코드를 직접 실행하거나, 그래프 이미지를 저장하여 보여주는 방식으로 변경해야 함
  - **가장 간단하게 그래프를 확인하는 방법은 에이전트가 작성한 코드를 복사해서 노트북 셀에서 실행하는 것**
  - 자동화: **에이전트가 코드를 실행한 후, 그 코드를 사용자에게 보여주도록 유도**
    - 이전의 사용자 정의 함수 = 도구의 **호출 정보(인자)** 만 출력
    - **실행된 코드를 출력하도록 유도 → 그래프 확인 가능**

```
from langchain_core.messages import AIMessage, ToolMessage
import json

def parse_agent_stream(stream):
    """
    LangGraph Agent의 스트림 출력을 파싱하여 깔끔하게 보여주는 함수
    """
    for chunk in stream:
        if "model" in chunk:
            model_messages = chunk["model"]["messages"]
            for message in model_messages:
                if isinstance(message, AIMessage):
                    if message.tool_calls:
                        for tool_call in message.tool_calls:
                            tool_name = tool_call['name']
                            tool_args = tool_call['args']

                            # tool_args가 문자열이면 딕셔너리로 변환 시도
                            if isinstance(tool_args, str):
                                try:
                                    tool_args = json.loads(tool_args)
                                except json.JSONDecodeError:
                                    pass # 변환 실패 시 그대로 둠

                            # 코드 실행 도구인 경우
                            if isinstance(tool_args, dict) and "code" in tool_
                                print(f"🛠️ [코드 생성]\n")
                                print("```python")
                                print(tool_args['code'])
                                print("```")
```

```
                            else:
                                print(f"🛠️ 도구 호출: {tool_name} (인자: {tool_ar

                    elif message.content:
                        print(f"🤖 AI 답변: {message.content}")

            elif "tools" in chunk:
                tool_messages = chunk["tools"]["messages"]
                for message in tool_messages:
                    if isinstance(message, ToolMessage):
                        print(f"✅ 실행 완료: {message.content}")
```

```
# 스트림 실행
result_stream = agent.stream(
    {"messages" : [HumanMessage(content="matplotlib 예제 코드 보여주고, 실제 파이차트를
    config={"recursion_limit": 10}
)

parse_agent_stream(result_stream)
```

- 셀 출력 (`2.2s`)
  - 🛠️ [코드 생성]

```
import matplotlib.pyplot as plt

# Pie chart data
labels = ['Apple', 'Banana', 'Cherry', 'Date']
sizes = [15, 30, 45, 10]

# Create pie chart
fig1, ax1 = plt.subplots()
ax1.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
ax1.axis('equal') # Equal aspect ratio ensures that pie is drawn as a ci

plt.show()
```

  - ✅ 실행 완료: ModuleNotFoundError("No module named 'matplotlib'")

- 사전에 `matplotlib` 설치 필요

```
%pip install matplotlib
```

```
# 스트림 실행
result_stream = agent.stream(
    {"messages" : [HumanMessage(content="matplotlib을 사용하여 파이 차트를 그리는 코드를
    config={"recursion_limit": 10}
)

parse_agent_stream(result_stream)
```

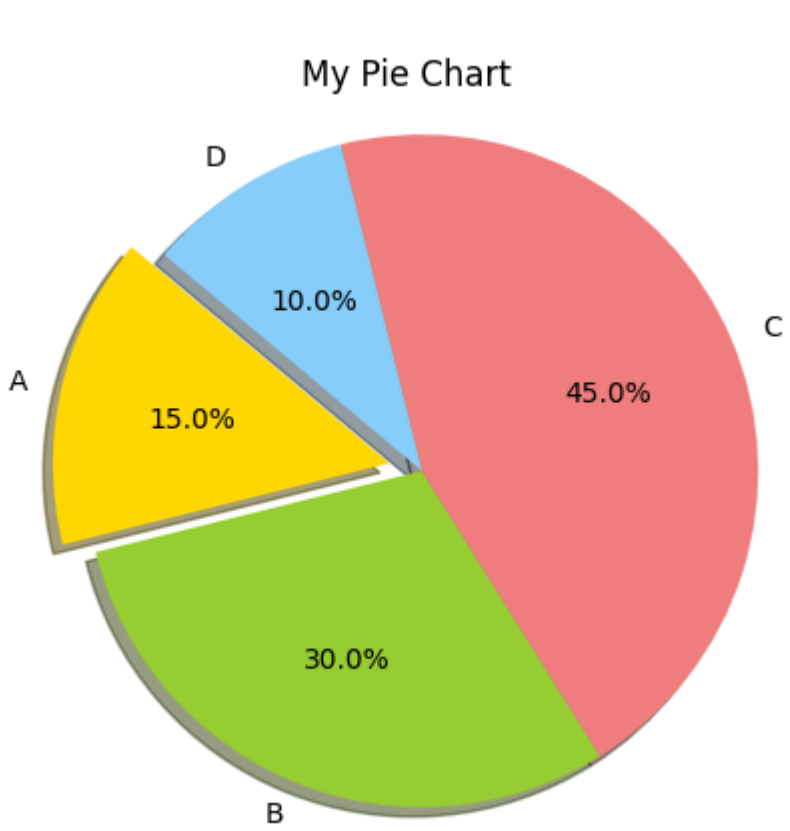
- 셀 출력 (`4.4s`)
  - 🛠️ [코드 생성]

```
import matplotlib.pyplot as plt

# Data for the pie chart
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
explode = (0.1, 0, 0, 0)  # explode 1st slice

# Plot
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
autopct='%1.1f%%', shadow=True, startangle=140)

plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a c
plt.title('My Pie Chart')
plt.show()
```



- ✅ 실행 완료:
- 🤖 AI 답변: 파이 차트가 성공적으로 생성되었습니다. (참고: `plt.show()`는 새 창에 차트를 표시하므로, 웹 기반 환경에서는 직접적인 시각적 출력을 볼 수 없을 수 있습니다.)

- next: ***`03. 에이전트(Agent)`*** ②
