

CH03 - 출력 파서(Output Parsers)

출력파서 (Output Parser)

: LangChain의 출력파서는 언어 모델(LLM)의 출력을 더 유용하고 구조화된 형태로 변환하는 중요한 도구

LangChain 프레임워크에서는 모델의 자유로운 텍스트 출력을 받아 JSON, 리스트, 딕셔너리 등 **일관된 형식으로 가공**할 수 있도록 도와준다.

I. PydanticOutputParser

개념

PydanticOutputParser는

LLM의 자유 텍스트 출력을 **Pydantic** 모델 기반의 구조화된 데이터로 변환해주는 LangChain의 출력 파서이다.

즉, 모델이 만든 텍스트 응답을 자동으로 검증된 JSON·객체 형태로 바꿔줌으로써 후속 데이터 처리에 바로 활용할 수 있다.

주요 구성 요소

1. **get_format_instructions()**

- LLM에게 **출력 형식(format)**을 지시하는 명령어를 제공한다.
- 프롬프트에 포함되어, 모델이 Pydantic 구조에 맞는 형태로 답변하도록 유도한다.

2. **parse()**

- LLM이 생성한 **문자열 출력을 받아**
 - 지정된 **Pydantic 모델 객체로 변환**하고
 - 필드 유효성 검사를 수행한다.

예시 코드

```
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field
from langchain.prompts import ChatPromptTemplate

# 1. Pydantic 모델 정의
class EmailSummary(BaseModel):
    person: str = Field(description="메일을 보낸 사람")
    email: str = Field(description="보낸 사람의 이메일 주소")
    subject: str = Field(description="메일 제목")
    summary: str = Field(description="메일 내용을 요약한 텍스트")
    date: str = Field(description="미팅 날짜와 시간")

# 2. 출력 파서 생성
parser = PydanticOutputParser(pydantic_object=EmailSummary)

# 3. 프롬프트 템플릿 생성
prompt = ChatPromptTemplate.from_template("""
아래 이메일을 요약하고 구조화하세요:
{email_text}
{format_instructions}
""")
```

```
# 4. 체인 구성
chain = prompt | model | parser
```

* `with_structured_output()` 으로 체인 생성

LangChain에서는 `PydanticOutputParser` 대신 `with_structured_output()` 메서드를 이용해 더 간결하게 구조화된 출력을 얻을 수 있다.

```
# with_structured_output() 사용 예시
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_openai import ChatOpenAI

# Pydantic 모델 정의
class EmailSummary(BaseModel):
    person: str = Field(description="메일 보낸 사람")
    email: str = Field(description="메일 주소")
    subject: str = Field(description="메일 제목")
    summary: str = Field(description="요약")
    date: str = Field(description="미팅 날짜 및 시간")

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

# parser가 자동으로 추가된 체인 생성
structured_llm = model.with_structured_output(EmailSummary)

# 체인 실행
result = structured_llm.invoke("아래 이메일을 요약해줘 ...")
print(result)
```

결과:

`result` 는 이미 `EmailSummary` 객체 형태로 반환된다
즉, `PydanticOutputParser` 를 별도로 지정하지 않아도 된다.

장점

항목	설명
자동 구조화	모델 응답을 즉시 Pydantic 객체로 변환
데이터 검증	필드 타입·형식 검증 자동 수행
간결성	<code>with_structured_output()</code> 으로 체인 생성이 단순
유연성	JSON, 딕셔너리, 객체 형태 모두 대응 가능

주의사항

- `with_structured_output()` 은 일부 스트리밍 기능 미지원
- 모델이 출력 형식을 지키지 않으면 **ValidationError** 발생 가능
- 프롬프트에 ****명확한 `format_instructions` ***를 포함하는 것이 중요

결론

- **PydanticOutputParser**는 LangChain에서 **LLM 출력을 구조화하고 검증**하는 핵심 도구이다.
- `with_structured_output()` 을 사용하면 **parser가 자동 내장된 체인 생성**이 가능해 더 간결하고 직관적인 개발이 가능하다.

II. 콤마 구분자 출력 파서 (CommaSeparatedListOutputParser)

개념

CommaSeparatedListOutputParser는

LLM의 출력을 **쉼표(,)로 구분된 항목 목록(List)** 형태로 변환할 때 사용하는 LangChain의 출력 파서이다.

여러 개의 데이터 포인트, 이름, 항목 등을 나열해야 할 때 유용하며,
가독성과 구조화된 결과를 동시에 제공합니다.

주요 목적

- 쉼표로 구분된 리스트 출력 자동 처리
- 데이터 정리 및 나열형 결과 생성에 최적화
- 간단한 프롬프트 지침만으로 구조화된 결과 확보

예:

“대한민국 관광명소 5가지를 나열하라.”

→ ['경복궁', '인사동', '부산 해운대해수욕장', '제주도', '남산타워']

전체 코드 예시

```
# 환경 설정
from dotenv import load_dotenv
load_dotenv()

# LangSmith 추적 설정
from langchain_teddynote import logging
logging.langsmith("CH03-OutputParser") # 프로젝트명 설정

# 필요한 모듈 import
from langchain_core.output_parsers import CommaSeparatedListOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

# 1. 콤마 구분 리스트 파서 초기화
output_parser = CommaSeparatedListOutputParser()

# 2. 출력 형식 지침 가져오기
format_instructions = output_parser.get_format_instructions()

# 3. 프롬프트 템플릿 정의
prompt = PromptTemplate(
    template="List five {subject}.\n{format_instructions}",
    input_variables=["subject"],
    partial_variables={"format_instructions": format_instructions},
)

# 4. 모델 초기화
model = ChatOpenAI(temperature=0)

# 5. 프롬프트 + 모델 + 출력 파서 연결 (체인 구성)
chain = prompt | model | output_parser
```

체인 실행

```
# "대한민국 관광명소"에 대한 결과 생성
result = chain.invoke({"subject": "대한민국 관광명소"})
print(result)
```

출력 결과

```
['경복궁', '인사동', '부산 해운대해수욕장', '제주도', '남산타워']
```

스트리밍 처리 (chain.stream)

```
# 스트리밍 모드로 결과 확인
for s in chain.stream({"subject": "대한민국 관광명소"}):
    print(s)
```

출력 결과 (스트리밍으로 순차 출력)

```
['경복궁']
['인사동']
['부산 해운대해수욕장']
['제주도']
['남산타워']출력 결과 (스트리밍으로 순차 출력)
```

스트리밍을 사용하면 한 항목씩 실시간 처리가 가능하며, UI에서 결과를 점진적으로 표시하거나 대화형 출력을 구성할 때 유용하다.

장점

항목	설명
단순성	추가 스키마 없이도 리스트 형태의 출력 확보
유연성	문자열 → 리스트 변환 자동 처리
스트리밍 지원	chain.stream()으로 실시간 처리 가능
명확한 형식	쉼표 기반 구분으로 사람이 읽기 쉬운 형태 제공

주의사항

- 모델이 ****형식 지침(`format_instructions`)****을 따르지 않으면 올바른 리스트가 반환되지 않을 수 있음
- 쉼표가 포함된 문장이나 데이터는 **잘못 파싱될 수 있으므로 주의 필요**
- 일반 텍스트가 아닌 **명확한 나열 요청(prompt)** 형태로 사용하는 것이 바람직함

결론

- CommaSeparatedListOutputParser**는 간단한 리스트 형태의 결과를 구조화하기에 가장 적합한 파서이다.
- Prompt + Model + OutputParser 체인**만으로 데이터를 명확하게 정리된 형태로 받을 수 있으며, **chain.stream()** 기능으로 스트리밍 출력까지 손쉽게 구현 가능하다.

III. 구조화된 출력 파서 (StructuredOutputParser)

개념

StructuredOutputParser는

LLM의 응답을 **key-value 형태의 딕셔너리(dict)** 로 구성하고, **여러 개의 필드(field)** 를 명확하게 구분하여 반환할 때 사용하는 출력 파서이다.

즉, 모델의 답변을 단순 텍스트가 아니라

`{ "answer": "...", "source": "..." }` 와 같은 **구조화된 데이터**로 변환한다.

Pydantic / JSON 파서와의 차이점

항목	StructuredOutputParser	PydanticOutputParser
특징	단순한 dict 구조 기반	엄격한 Pydantic 검증 기반
적용 대상	가벼운 / 로컬 모델	GPT·Claude 등 고성능 모델
검증 수준	낮음 (형식 중심)	높음 (타입·값 검증)
활용 사례	단순 key-value 데이터	복잡한 스키마·유효성 검사

참고:

로컬 모델은 `PydanticOutputParser` 가 동작하지 않는 경우가 많으므로,

대안으로 `StructuredOutputParser` 를 사용하는 것이 좋다.

🔧 전체 코드 예시

```
# 환경 설정
from dotenv import load_dotenv
load_dotenv()

# LangSmith 추적 설정
from langchain_teddynote import logging
logging.langsmith("CH03-OutputParser") # 프로젝트명 설정

# 필요한 모듈 import
from langchain.output_parsers import ResponseSchema, StructuredOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

# 1. 응답 스키마 정의
response_schemas = [
    ResponseSchema(name="answer", description="사용자의 질문에 대한 답변"),
    ResponseSchema(
        name="source",
        description="질문에 답하기 위해 사용된 출처(웹사이트 주소)",
    ),
]

# 2. 구조화된 출력 파서 초기화
output_parser = StructuredOutputParser.from_response_schemas(response_schemas)

# 3. 형식 지시사항 가져오기
format_instructions = output_parser.get_format_instructions()

# 4. 프롬프트 템플릿 생성
prompt = PromptTemplate(
    template=(
        "answer the user's question as best as possible.\n"
        "{format_instructions}\n{question}"
    ),
```

```

input_variables=["question"],
partial_variables={"format_instructions": format_instructions},
)

# 5. 모델 및 체인 생성
model = ChatOpenAI(temperature=0)
chain = prompt | model | output_parser

```

체인 실행 예시

```

# 질문 실행
result = chain.invoke({"question": "대한민국의 수도는 어디인가요?"})
print(result)

```

출력 결과

```
{'answer': '서울', 'source': 'https://ko.wikipedia.org/wiki/서울'}
```

스트리밍 처리 (chain.stream)

```

for s in chain.stream({"question": "세종대왕의 업적은 무엇인가요?}):
    print(s)

```

출력 결과 (실시간 스트림 형태)

```
{'answer': '세종대왕은 한글을 창제하고 문화를 발전시켰습니다.',
'source': 'https://ko.wikipedia.org/wiki/세종대왕'}
```

스트리밍의 장점: 모델의 출력을 한 번에 받는 대신, **순차적으로 수신하며 실시간 피드백**을 받을 수 있다.

장점

항목	설명
단순 구조화	key-value 기반으로 결과를 쉽게 파싱
가벼운 모델 호환성	로컬 LLM에서도 잘 작동
유연한 형식 지정	ResponseSchema로 원하는 필드 자유롭게 정의
빠른 처리	복잡한 검증 없이 즉시 변환 가능

주의사항

- PydanticOutputParser에 비해 **검증 강도가 약함**
- LLM이 형식 지침을 지키지 않으면 key 누락 가능
- ResponseSchema 이름과 설명을 명확히** 지정해야 정확한 결과 반환

결론

- StructuredOutputParser**는 LLM의 답변을 **간단한 dict 형태로 구조화**하기 위한 도구이다.
- 특히 **로컬 모델이나 경량화 모델**을 사용할 때, **PydanticOutputParser의 대체재**로 이상적이다.

GPT·Claude 등 대형 모델에는 PydanticOutputParser,
로컬·소형 모델에는 StructuredOutputParser를 사용하는 것이 좋다.

IV. JSON 출력 파서 (JsonOutputParser)

개념

JsonOutputParser는

LLM이 생성한 출력을 **사용자가 정의한 JSON 구조(JSON Schema)**에 맞게 변환해주는 LangChain의 출력 파서이다.

즉, LLM이 생성하는 자유 텍스트 대신 **정확한 JSON 객체 형태로 데이터를** 받을 수 있도록 한다.

특징

항목	설명
명시적 스키마 지정	사용자가 원하는 JSON 구조를 직접 정의 가능
Pydantic 연동 지원	BaseModel 기반으로 JSON 필드 자동 검증
모델 용량 의존성	복잡한 JSON을 정확히 생성하려면 대형 모델(GPT·Claude 등)이 필요
유연한 사용	Pydantic 기반 / 비기반 두 가지 방식 모두 지원

참고:

소형 또는 로컬 모델(예: **Llama-8B**)은 JSON 구조를 정확히 따르지 못할 수 있다.

이런 경우 간단한 파서를 사용하는 것이 더 안정적이다.

전체 코드 예시 (Pydantic 사용)

```
# 환경 설정
from dotenv import load_dotenv
load_dotenv()

# LangSmith 추적 설정
from langchain_teddynote import logging
logging.langsmith("CH03-OutputParser")

# 필요한 모듈 import
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import JsonOutputParser
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI

# 모델 초기화
model = ChatOpenAI(temperature=0, model_name="gpt-4o")

# 1. 원하는 JSON 데이터 구조 정의
class Topic(BaseModel):
    description: str = Field(description="주제에 대한 간결한 설명")
    hashtags: str = Field(description="해시태그 형식의 키워드(2개 이상)")

# 2. JsonOutputParser 초기화
parser = JsonOutputParser(pydantic_object=Topic)

# 3. 프롬프트 구성
prompt = ChatPromptTemplate.from_messages([
    ("system", "당신은 친절한 AI 어시스턴트입니다. 질문에 간결하게 답변하세요."),
    ("user", "#Format: {format_instructions}\n\n#Question: {question}")
])
```

```
# 4. format 지시사항 삽입
prompt = prompt.partial(format_instructions=parser.get_format_instructions())

# 5. 체인 구성
chain = prompt | model | parser
```

체인 실행

```
# 질의 실행
question = "지구 온난화의 심각성 대해 알려주세요."
result = chain.invoke({"question": question})
print(result)
```

출력 결과

```
{
  'description': '지구 온난화는 인간 활동으로 인한 온실가스 배출로 지구 평균 기온이 상승하는 현상입니다. 이로 인해 극지방 빙하가 녹고 해수면이 상승하며 기후 변화가 가속화됩니다.',
  'hashtags': '#지구온난화 #기후변화 #온실가스'
}
```

Pydantic 없이 사용하는 방법

`JsonOutputParser` 는 **Pydantic 없이도** 사용할 수 있다.

이 경우 스키마는 지정하지 않지만, **모델이 JSON 형식으로 응답하도록 유도한다.**

```
# 질의 작성
question = "지구 온난화에 대해 알려주세요. 설명은 `description`, 관련 키워드는 `hashtags`에 도와주세요."

# JsonOutputParser 초기화 (Pydantic 미사용)
parser = JsonOutputParser()

# 프롬프트 설정
prompt = ChatPromptTemplate.from_messages([
    ("system", "당신은 친절한 AI 어시스턴트입니다. 질문에 간결하게 답변하세요."),
    ("user", "#Format: {format_instructions}\n\n#Question: {question}")
])

# format 지시사항 삽입
prompt = prompt.partial(format_instructions=parser.get_format_instructions())

# 체인 구성 및 실행
chain = prompt | model | parser
response = chain.invoke({"question": question})
print(response)
```

출력 결과

```
{
  'description': '지구 온난화는 대기 중 온실가스 농도의 증가로 인해 지구의 평균 기온이 상승하는 현상을 말합니다. 이는 기후 변화, 해수면 상승, 극지방 빙하의 감소 등 다양한 환경 문제를 초래합니다.',
}
```



```
'hashtags': ['#지구온난화', '#기후변화', '#온실가스', '#해수면상승', '#환경문제']
}
```

장점

항목	설명
정확한 구조화	모델 출력이 JSON 형태로 보장됨
검증 가능	Pydantic을 이용한 데이터 유효성 검사
유연성	Pydantic 사용 여부에 따라 다양한 시나리오 적용
표준 포맷화	JSON 기반 응답으로 API·DB 연동 용이

주의사항

- 모델의 **용량(Intelligence Level)** 이 충분하지 않으면 JSON 형식을 깨뜨릴 수 있음
- `format_instructions` 를 명확히 포함하지 않으면 올바른 JSON이 생성되지 않을 수 있음
- 해시태그나 리스트 등 복잡한 데이터 구조는 대형 모델(GPT-4급) 권장

결론

- **JsonOutputParser**는 LangChain에서 LLM의 응답을 **정확한 JSON 형식으로 구조화**할 수 있는 강력한 도구이다.
- **Pydantic 기반**으로 정의하면 필드 검증까지 자동 수행되고,
- **비기반 모드**로 사용하면 더 단순하게 JSON 출력을 받을 수 있다.

정리:

- 고정된 스키마 → `JsonOutputParser(pydantic_object=Model)`
- 자유 JSON → `JsonOutputParser()`

V. 데이터프레임 출력 파서(PandasDataFrameOutputParser)

개념

PandasDataFrameOutputParser는

LLM의 출력을 **Pandas DataFrame** 형태로 변환하는 LangChain의 출력 파서이다.

즉, 모델이 생성한 표 형태의 텍스트 결과를

자동으로 `pd.DataFrame` 객체로 변환해 준다.

주요 기능

기능	설명
데이터 구조화	LLM의 텍스트 결과 → DataFrame으로 변환
직관적 조작 가능	이후 Pandas 연산(mean, sum, groupby 등)에 바로 활용
결과 검증 용이	구조화된 형태라 시각화·검증이 간편
RetryOutputParser 와 연동 가능	잘못된 형식일 경우 자동 재시도 가능

간단 예시

```
# 쿼리 입력
df_query = "Calculate average `Fare` rate."
```

```
# 체인 실행
parser_output = chain.invoke({"query": df_query})

print(parser_output)
# 출력 예시
{'mean': 22.19937}

# 실제 검증
df["Fare"].mean() # 22.19937
```

LLM이 반환한 결과를 DataFrame 포맷으로 파싱해
Pandas 연산(`mean()` , `sum()` , `describe()`) 등에 바로 활용 가능.

핵심 요약

- 목적: LLM 결과를 표 형태로 구조화
- 형식: `pandas.DataFrame` 으로 자동 변환
- 활용: 데이터 요약, 통계, 시각화, 검증 등
- 조합: `RetryOutputParser` 와 함께 사용 시 안정성 향상

한 줄 정리:

PandasDataFrameOutputParser는

LLM의 출력 결과를 즉시 DataFrame으로 변환해 데이터 분석에 활용할 수 있게 해준다.

VI. 날짜 형식 출력 파서 (DatetimeOutputParser)

개념

DatetimeOutputParser는

LLM의 출력을 Python `datetime` 형식으로 자동 파싱해주는 LangChain의 출력 파서이다.

즉, LLM이 반환한 날짜·시간 정보를

일정한 포맷(`%Y-%m-%d` , `%H:%M:%S` 등) 으로 변환하여 처리할 수 있다.

주요 기능

기능	설명
자동 날짜 파싱	LLM이 생성한 문자열 → <code>datetime</code> 객체로 변환
출력 형식 지정 가능	<code>output_parser.format = "%Y-%m-%d"</code> 등으로 자유롭게 설정
유연한 포맷 지원	Python <code>strftime()</code> 포맷 코드 사용 가능
체인 결합 용이	Prompt + LLM + Parser 형태로 바로 연결 가능

간단 예시

```
from langchain.output_parsers import DatetimeOutputParser
from langchain.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
```

```
# 날짜 포맷 파서 설정
output_parser = DatetimeOutputParser()
output_parser.format = "%Y-%m-%d"

# 프롬프트 템플릿
template = ""Answer the users question:

#Format Instructions:
{format_instructions}

#Question:
{question}

#Answer:"""

prompt = PromptTemplate.from_template(
    template,
    partial_variables={
        "format_instructions": output_parser.get_format_instructions()
    },
)

# 체인 구성
chain = prompt | ChatOpenAI() | output_parser

# 실행
output = chain.invoke({"question": "Google 이 창업한 연도는?"})
print(output.strftime("%Y-%m-%d"))
# → '1998-09-04'
```

주요 형식 코드 요약

코드	의미	예시
%Y	4자리 연도	2024
%m	월(2자리)	07
%d	일(2자리)	04
%H	시(24시간제)	14
%M	분	45
%S	초	08
%p	AM/PM	PM

예시 포맷:

"%Y-%m-%d" → 2024-07-04

"%Y/%m/%d %H:%M" → 2024/07/04 14:45

핵심 요약

- **목적:** LLM의 출력 문자열을 **datetime 객체로 자동 변환**
- **활용:** 날짜 계산, 비교, 로그 분석 등
- **장점:** 일관된 포맷 유지로 후속 연산 및 정렬에 유리

한 줄 정리:

DatetimeOutputParser는

LLM의 문자열 출력을 **표준 날짜 형식(datetime)** 으로 정확히 변환해주는 파서다.

VII. 열거형 출력 파서 (EnumOutputParser)

개념

EnumOutputParser는

LLM의 출력을 **Enum(열거형)** 값으로 파싱해주는 LangChain의 출력 파서이다.

즉, 모델이 "빨간색" 같은 일반 텍스트를 반환하더라도 이를 자동으로 정의된 **Enum 객체(Colors.RED)** 로 변환해 준다.

주요 기능

기능	설명
Enum 매핑	LLM의 문자열을 Enum 값으로 변환
정확한 검증	지정된 Enum 외의 값이 오면 예외 발생
체인 결합	Prompt + LLM + Parser 형태로 간단히 연결
유연한 응답 처리	Enum 멤버로 반환되어 코드 내 제어문 사용 용이

전체 예시 코드

```
# 1. Enum 클래스 정의
from enum import Enum

class Colors(Enum):
    RED = "빨간색"
    GREEN = "초록색"
    BLUE = "파란색"

# 2. EnumOutputParser 생성
from langchain.output_parsers.enum import EnumOutputParser
parser = EnumOutputParser(enum=Colors)

# 3. 프롬프트 + 모델 + 파서 연결
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

prompt = PromptTemplate.from_template(
    """다음의 물체는 어떤 색깔인가요?

Object: {object}

Instructions: {instructions}"""
).partial(instructions=parser.get_format_instructions())

# ChatOpenAI 모델 생성
model = ChatOpenAI(temperature=0)
```

```
# 프롬프트, 모델, 파서를 연결한 체인 생성
chain = prompt | model | parser
```

체인 실행 (`chain.invoke()`)

```
# "하늘"에 대한 색상 판별 요청
response = chain.invoke({"object": "하늘"})

print(response)
# 출력 결과 → Colors.BLUE
print(response.value)
# 출력 결과 → '파란색'
```

설명:

- LLM은 “파란색”이라는 문자열을 생성
- `EnumOutputParser` 가 이를 `Colors.BLUE` 멤버로 변환
- 따라서 코드에서는 `response` 를 `Enum` 객체로, `response.value` 를 원래 문자열 값으로 사용할 수 있다.

핵심 요약

- 목적: 모델 출력을 미리 정의된 Enum으로 변환
- 출력 타입: `Enum` 객체 (예: `Colors.BLUE`)
- 활용: 상태값, 감정분류, 색상판단, 옵션선택 등 고정된 선택지 상황

활용 예시

Enum 이름	값	활용 사례
<code>Sentiment(Enum)</code>	POSITIVE / NEGATIVE / NEUTRAL	감정 분석
<code>TaskStatus(Enum)</code>	TODO / DOING / DONE	작업 상태 추적
<code>Weather(Enum)</code>	SUNNY / CLOUDY / RAINY	날씨 분류

한 줄 정리:

`EnumOutputParser` 는 LLM의 자유 텍스트 출력을 미리 정의된 Enum 타입으로 정확히 변환해주는 파서다.

VIII. 출력 수정 파서 (OutputFixingParser)

개념

`OutputFixingParser`는

LLM 출력이 스키마나 형식에 맞지 않을 때, 자동으로 형식을 수정하여 재파싱(re-parse) 하는 LangChain의 파서이다.

즉, 기존 파서(PydanticOutputParser 등)를 감싸서 잘못된 출력이 감지되면 LLM에게 수정 요청을 자동으로 재전송한다.

작동 원리

단계	설명
①	LLM이 첫 번째 응답을 생성
②	내부 파서(Pydantic 등)가 형식 검증 수행

단계	설명
③	형식 불일치 발생 시, OutputFixingParser가 감지
④	오류 내용을 LLM에게 전달하며 수정 요청
⑤	LLM이 올바른 형식으로 다시 응답 생성

핵심 아이디어:

“형식이 틀리면 자동으로 다시 시켜라.”

간단 예시

```
from dotenv import load_dotenv
load_dotenv()

from langchain_openai import ChatOpenAI
from langchain.output_parsers import PydanticOutputParser, OutputFixingParser
from langchain_core.pydantic_v1 import BaseModel, Field
from typing import List
```

```
# 1. 데이터 스키마 정의
class Actor(BaseModel):
    name: str = Field(description="name of an actor")
    film_names: List[str] = Field(description="list of films they starred in")
```

```
# 2. 기본 파서 (PydanticOutputParser)
parser = PydanticOutputParser(pydantic_object=Actor)

# 잘못된 형식의 데이터 예시 (따옴표가 JSON 표준에 맞지 않음)
misformatted = '{"name': 'Tom Hanks', 'film_names': ['Forrest Gump']}'

# 기본 파서로 시도 → 오류 발생
parser.parse(misformatted)
# ❌ ValidationError / JSONDecodeError 발생
```

OutputFixingParser로 자동 수정

```
# 3. OutputFixingParser 래핑
new_parser = OutputFixingParser.from_llm(parser=parser, llm=ChatOpenAI())

# 4. 잘못된 형식을 자동 수정
actor = new_parser.parse(misformatted)

print(actor)
# 출력 결과 → Actor(name='Tom Hanks', film_names=['Forrest Gump'])
```

LLM이 내부적으로 '작은따옴표' → '큰따옴표' 수정 후, 올바른 JSON으로 변환하여 재파싱 완료.

핵심 요약

항목	설명
역할	잘못된 형식의 LLM 출력 자동 수정
구조	다른 파서(Pydantic, JSON 등)를 감싸서 동작
자동 재시도	오류 감지 → 수정 요청 → 재파싱

항목	설명
출력 보정	작은 문법 오류나 필드 누락 자동 보정 가능

주의사항

- LLM 추가 호출이 발생하므로 응답 지연 가능
- 단순 형식 오류 수정용으로 적합 (논리 오류는 수정 불가)
- 내부적으로 사용하는 LLM의 품질이 낮으면 수정 정확도도 낮아짐

결론

- **OutputFixingParser**는 LLM이 생성한 잘못된 형식을 **자동으로 수정하고 올바르게 재파싱**하는 안전장치 역할을 한다.
- 특히 **PydanticOutputParser**와 함께 사용하면 형식 안정성과 신뢰성이 크게 향상된다.

한 줄 정리:

“LLM이 틀려도 괜찮아 — **OutputFixingParser**가 알아서 고쳐준다.”

IX. LangChain Output Parser 총정리

번호	파서 이름	주요 역할	입력 예시	출력 형태	사용 시점 / 특징
01	PydanticOutputParser	LLM 출력을 Pydantic 모델 기반 구조화	텍스트 → JSON 형태	BaseModel 객체 (검증됨)	가장 강력한 구조화 방식 → 복잡한 데이터 검증 가능
02	CommaSeparatedListOutputParser	쉼표(,)로 구분된 리스트로 변환	"사과, 바나나, 포도"	['사과', '바나나', '포도']	단순 리스트나 키워드 나열에 적합 스트리밍 처리 지원
03	StructuredOutputParser	간단한 dict(key-value) 형식으로 파싱	"answer: 서울, source: 위키백과"	{'answer': '서울', 'source': '위키백과'}	로컬/경량 모델용 Pydantic보다 단순
04	JsonOutputParser	원하는 JSON 스키마 기반 출력 생성	"description: ..., hashtags: ..."	JSON 객체 또는 Pydantic 모델	JSON 구조 명시 가능 모델 인텔리전스 필요
05	PandasDataFrameOutputParser	LLM 출력을 Pandas DataFrame 형태로 변환	"평균 요금 계산"	pd.DataFrame or dict → DataFrame	데이터 분석/통계에 활용 Pandas 연산과 바로 연동
06	DatetimeOutputParser	날짜/시간 문자열 → datetime 객체 변환	"1998-09-04"	datetime.datetime(1998, 9, 4)	일자, 시간 계산에 사용 strftime 포맷 지원
07	EnumOutputParser	LLM 출력을 Enum 값 으로 변환	"파란색"	Colors.BLUE	고정된 선택지 분류(색상, 상태, 감정 등)에 적합
08	OutputFixingParser	출력 오류 자동 수정 + 재파싱	잘못된 JSON, 따옴표 불일치 등	올바르게 수정된 결과	다른 파서를 감싸 자동 보정 “LLM이 틀려도 고쳐줌”