Ch 03. Output Parser

상태	완료
최종 편집 일시	@2025년 11월 8일 오후 5:49

https://wikidocs.net/233771

출력파서(Output Parser)

: LangChain의 출력파서는 언어 모델(LLM)의 **출력**을 더 유용하고 구조화된 형태로 변환하는 중요한 컴포넌트

→ 지금까지는 입력에 신경썼다면 이번 챕터에서는 출력 양식에 집중한다.

출력파서의 역할

- LLM의 출력을 변환한다.

  > (다시말해 LLM의 출력 성능에는 영향을 끼치지 않고 종합 및 정리에 초점을 맞춘다.)

- **구조화된 데이터 생성**에 매우 유용

  > 양식을 지정하니 당연한 말

- LangChain 프레임워크에서 **다양한 종류의 출력 데이터**를 파싱하고 처리

  > 양식을 다양하게 정할 수 있으니 맞는 말

출력파서의 이점

1. **구조화**: LLM의 자유 형식 텍스트 출력을 **구조화된 데이터**로 변환합니다.
2. **일관성**: 출력 형식을 일관되게 유지하여 **후속 처리를 용이**하게 합니다.
3. **유연성**: 다양한 출력 형식(JSON, 리스트, 딕셔너리 등)으로 변환이 가능합니다.

---

01. Pydantic 출력 파서(PydanticOutputParser)

`PydanticOutputParser` 의 핵심 메서드

- `get_format_instructions()` : 언어 모델이 출력해야 할 정보의 형식 (데이터의 필드와 형태)을 정의하는 지침(instruction) 을 제공
- `parse()` : 언어 모델의 출력(문자열로 가정)을 받아들여 이를 특정 구조로 분석하고 변환 (실제 parsing을 실행하는 메소드)

💡 Pydantic이 무엇인가?
https://docs.pydantic.dev/latest/

> **Data validation library** for Python.

- Python에서 데이터 검증 및 데이터 범위 설정 등 데이터를 효과적이고 빠르게 관리하기 위해 사용되는 라이브러리

```
from datetime import datetime

import logfire

from pydantic import BaseModel

logfire.configure()
logfire.instrument_pydantic()


class Delivery(BaseModel):
    timestamp: datetime
    dimensions: tuple[int, int]


# this will record details of a successful validation to logfire
m = Delivery(timestamp='2020-01-02T03:04:05Z', dimensions=
['10', '20'])
print(repr(m.timestamp))
#> datetime.datetime(2020, 1, 2, 3, 4, 5, tzinfo=TzInfo(UTC))
print(m.dimensions)
#> (10, 20)

Delivery(timestamp='2020-01-02T03:04:05Z', dimensions=['10'])
```

- Pydantic 파서를 활용해 Langchain의 요약된 내용을 사전에 정의된 클래스를 활용하여 구조화된 정보로 정리할 수 있게 된다.

```
class EmailSummary(BaseModel):
    person: str = Field(description="메일을 보낸 사람")
    email: str = Field(description="메일을 보낸 사람의 이메일 주소")
    subject: str = Field(description="메일 제목")
    summary: str = Field(description="메일 본문을 요약한 텍스트")
    date: str = Field(description="메일 본문에 언급된 미팅 날짜와 시간")


# PydanticOutputParser 생성
parser = PydanticOutputParser(pydantic_object=EmailSummary)
```

```
EmailSummary(
person='김철수',
email='chulsoo.kim@bikecorporation.me',
subject='"ZENESIS" 자전거 유통 협력 및 미팅 일정 제안',
summary='김철수 상무는 바이크코퍼레이션에서 ZENESIS 자전거의 브로슈어를 요청하며, 기술 사양, 배터리 성능, 디자인 정보가 필요하다고 언급했습니다. 또한, 협력 가능성을 논의하기 위해 1월 15일 화요일 오전 10시에 미팅을 제안했습니다.',
date='1월 15일 화요일 오전 10시')
```

- `.with_structured_output(Pydantic)`을 사용하여 출력 파서를 추가하면, 출력을 Pydantic 객체로 변환할 수 있다.
  - 기존에는 직접 체인에 parser를 추가해서 진행해야 했지만 이 메소드를 활용하면 llm 자체에 파서를 달 수도 있다.

    ```
    # 출력 파서를 추가하여 전체 체인을 재구성합니다.
    chain = prompt | llm | parser
    ```

    ```
    # llm에 직접 파서 달기
    llm_with_structered = ChatOpenAI(
        temperature=0, model_name="gpt-4o"
    ).with_structured_output(EmailSummary)

    # invoke() 함수를 호출하여 결과를 출력합니다.
    ```

```
answer = llm_with_structered.invoke(email_conversation)
answer
```

02. 콤마 구분자 출력 파서 (CommaSeparatedListOutputParser)

> `CommaSeparatedListOutputParser` 는 쉼표로 구분된 항목 목록을 반환한다.

```
# 체인 호출 실행
chain.invoke({"subject": "역대 F1 드라이버 중 최고의 드라이버"})
✓ [21] 1s 177ms

['Juan Manuel Fangio',
 'Ayrton Senna',
 'Michael Schumacher',
 'Lewis Hamilton',
 'Alain Prost']
```

List 형태로 값을 받아올 수 있다는 이점이 있다.→ CSV 파일과 연계하여 자동 저장 및 분류도 가능할 듯

03. 구조화된 출력 파서 (StructuredOutputParser)

> LLM에 대한 답변을 `dict` 형식으로 구성하고 key/value 쌍으로 갖는 여러 필드를 반환한다.

> Pydantic/JSON 파서가 더 강력하지만, 이는 덜 강력한 모델(예를 들어 로컬모델과 같은 인텔리전스가 GPT, Claude 모델보다 인텔리전스가 낮은(parameter 수가 낮은) 모델)에 유용
>
> → 왜 더 유용할까?

1. 애초에 로컬 모델에서는 Pydantic 파서가 작동하지 않을 때도 많다. (그때의 대안)

   > 어떤 경우 pydantic 파서가 작동하지 않을까?
   >
   > **1) 출력 일관성 부족할 때**
   >
   > - Pydantic 파서는 모델이 **정확히 JSON 스키마에 맞는 문자열**을 생성해야 정상 작동

> - 하지만 로컬 모델은 **언어 생성 품질이 낮아**, 괄호나 따옴표가 빠지거나, 포맷이 미묘하게 틀린 JSON을 자주 출력한다. 이 경우 `json.loads()` 단계에서 오류가 발생하여 파서가 실패한다 → `StructuredOutputParser`는 입력된 정보의 형식에 더 유연하다!
>
> **2) Validation 스키마가 복잡할 때**
>
> - Pydantic은 타입 검사(type validation)와 필드 제약조건(validation constraints)을 강하게 적용하지만, 로컬 모델은 이런 **정교한 필드 구조**를 이해하지 못하거나, 부정확하게 채워 넣는다.
> - `List[Dict[str, Union[int, str]]]` 같은 구조를 정확히 맞추지 못해 파서에 값을 넣을 경우 오류가 날 가능성이 높다.
>
> **3) 출력 제어 능력 부족**
>
> - GPT-4, Claude-3 같은 고성능 모델은 "JSON 형식으로 출력하라"는 명령을 잘 따르지만, 파라미터 수가 적은 로컬 모델(LLaMA, Mistral 등)은 이런 **명시적 형식 제어 능력(format adherence)** 이 약하다.

→ 요약하면 로컬 모델은 출력의 일관성, 복잡한 구조에 대한 제어 능력이 부족하기 때문에, Pydantic parser와 JSON parser와 같이 일관적인 입력값이 필요한 경우 오류를 일으킬 수 있다.

2. StructuredOutputParser는 단순한 문자열 분리와 매칭만 수행하므로 **리소스 사용량이 적고 빠르다.**

---

04. JSON 출력 파서(JsonOutputParser)

> 사용자가 원하는 **JSON 스키마**를 지정할 수 있게 해주며, 그 스키마에 맞게 LLM에서 데이터를 조회하여 결과를 도출
>
> → 다른 JSON 스키마 기반 분석 도구를 활용하기 좋아보임.

데이터 구조 정의에는 Pydantic을 활용한다.

```
# 원하는 데이터 구조를 정의합니다.
class Topic(pydantic.BaseModel):
    description: str = Field(description="주제에 대한 간결한 설명")
    hashtags: str = Field(description="해시태그 형식의 키워드(2개 이상)")
```

```
{'description': '상파울루 그랑프리 스프린트 레이스에서 맥스 페르스타펜이 우승할 가능성이 높습니다. 메르세데스와 페라리도 강력한 경쟁자로 예상됩니다
.',
 'hashtags': '#F1 #상파울루그랑프리 #스프린트레이스'}
```

물론 Pydantic 없이도 가능은 하다.

→ 이 경우 JSON을 반환하도록 요청하지만, 스키마에 대한 구체적인 정보는 설정할 수 없다.

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

# JSON 출력 파서 초기화
parser = JsonOutputParser()

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

# 지시사항을 프롬프트에 주입합니다.
prompt = prompt.partial(format_instructions=parser.get_format_instructions())

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

# 체인을 호출하여 쿼리 실행
response = chain.invoke({"question": question})

# 출력을 확인합니다.
print(response)
```

```
{'description': '상파울루 그랑프리 스프린트 레이스에서 맥스 페르스타펜이 우승할 것으로 예상됩니다. 그의 강력한 퍼포먼스와 레드불의 전략이 주요 요인입니다. 메르세데스와 페라리도 경쟁할 것으로 보이며, 날씨 변수도 결과에 영향을 미칠 수 있습니다.', 'hashtags': ['#F1', '#상파울루그랑프리', '#스프린트레이스', '#맥스페르스타펜', '#레드불', '#메르세데스', '#페라리']}
```

05. 데이터프레임 출력 파서 (PandasDataFrameOutputParser)

> 사용자가 임의의 `Pandas DataFrame`을 지정하고 해당 `DataFrame`에서 데이터를 추출하여 형식화된 사전 형태로 데이터를 조회할 수 있는 LLM을 요청할 수 있게 해준다.

- 사전에 정의된 `DataFrame` 을 통해 파서의 지시사항에서 양식을 설정할 수 있다.

```
# 출력 목적으로만 사용됩니다.
def format_parser_output(parser_output: Dict[str, Any]) -> None:
    # 파서 출력의 키들을 순회합니다.
    for key in parser_output.keys():
        # 각 키의 값을 딕셔너리로 변환합니다.
        parser_output[key] = parser_output[key].to_dict()
    # 예쁘게 출력합니다.
    return pprint.PrettyPrinter(width=4, compact=True).pprint(parser_output)

# 원하는 Pandas DataFrame을 정의합니다.
df = pd.read_csv("./data/titanic.csv")
df.head()

# 파서를 설정하고 프롬프트 템플릿에 지시사항을 주입합니다.
parser = PandasDataFrameOutputParser(dataframe=df)

# 파서의 지시사항을 출력합니다.
print(parser.get_format_instructions())
```

- Query를 이용하여 데이터 프레임에서의 통계를 낼 수도 있다.

```
# 임의의 Pandas DataFrame 작업 예시, 행의 수를 제한합니다.
df_query = "Retrieve the average of the Ages from row 0 to 4."

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

# 결과 출력
print(parser_output)
```

```
✓ [51] 1s 307ms
{'mean': 31.2}
```

- df 자체의 메소드로 할 수도 있는걸 왜 굳이 query로 찾는 법에 대해 알려주었는가?
  - LLM을 통해 자연어로 DataFrame을 다루는 방식에 대해 알 수 있다.
    (자연어→구조화 출력→데이터 매핑)
  - 다시 말해, 코드를 수정하지 않고 query를 다루는 방식으로 쉽게 데이터 매핑을 할 수 있게 된다.
- 예시에서 언급된 "잘못 형식화된 쿼리의 예시"의 의미

```
# 잘못 형식화된 쿼리의 예시입니다.
df_query = "Calculate average `Fare` rate."

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

# 결과 출력
print(parser_output)
```

  - **LLM 파서가 해석하기에 비정형적인 쿼리이므로 '형식상 잘못된'** 예시 → 구조화된 쿼리가 아니라고 하며 (비정형 문장이 아니다) 문제 제기를 하지만 잘 모르겠음.

---

06. 날짜 형식 출력 파서 (DatetimeOutputParser)

> `DatetimeOutputParser` 는 LLM의 출력을 `datetime` 형식으로 파싱하는 데 사용

형식 코드	설명	예시
%Y	4자리 연도	2024
%y	2자리 연도	24
%m	2자리 월	07
%d	2자리 일	04
%H	24시간제 시간	14
%I	12시간제 시간	02
%p	AM 또는 PM	PM
%M	2자리 분	45
%S	2자리 초	08

형식 코드	설명	예시
%f	마이크로초 (6자리)	000123
%z	UTC 오프셋	+0900
%Z	시간대 이름	KST
%a	요일 약어	Thu
%A	요일 전체	Thursday
%b	월 약어	Jul
%B	월 전체	July
%c	전체 날짜와 시간	Thu Jul 4 14:45:08 2024
%x	전체 날짜	07/04/24
%X	전체 시간	14:45:08

07. 열거형 출력 파서(EnumOutputParser)

> `EnumOutputParser` 는 `enum` 모듈을 사용하여 사전에 답변을 열거한뒤 답변에 해당하는 attribute로 설정가능하다.

`Enum` 자료형의 개념과 활용 의의
`Enum` (열거형, *Enumeration*)은 **정해진 값 집합 중 하나만 선택할 수 있게 하는 자료형**

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

parser = EnumOutputParser(enum=Colors)
```

→ 값의 **범위를 제한**하고 **의미 있는 이름(label)** 을 붙여준다 (일관성 있는 최종 출력 가능)
EnumOutputParser는 이 개념을 이용해 LLM의 출력이 **미리 정의한 Enum 값 중 하나인지 검사한다.**

- 답변이 미리 설정된 선택지중에 있었는데도 없다는 에러가 뜬다

```
response = chain.invoke({"object": "원자"})  # "원자" 에 대한 체인 호출 실행
print(response)

[62] 850ms
Traceback…
OutputParserException: Response '원자는 초록색입니다.' is not one of the expected values: ['빨간색', '초록색', '파란색']
For troubleshooting, visit: https://python.langchain.com/docs/troubleshooting/errors/OUTPUT_PARSING_FAILURE
```

- 아무것도 수정하지 않고 다시 돌리니 에러가 안 뜬다 왜 이런 현상이 발생할까

```
response = chain.invoke({"object": "원자"})  # "원자" 에 대한 체인 호출 실행
print(response)

[64] 759ms
Colors.RED
```

💡 실행에 따라 다른 결과값이 출력된 이유

- **현상**:
  첫 실행에서는 `"원자는 초록색입니다."`로 오류 발생
  → 두 번째 실행에서는 정상(`Colors.RED`).
- **원인**:
  1. LLM이 `"초록색"` 대신 문장 전체(`"초록색입니다."`)를 출력해 **Enum 매칭 실패**

     > EnumOutputParser는 정확하게 사전에 정의된 답변을 받아야 한다.

  2. LLM 출력의 **비결정성** (temperature > 0) → 그때그때 조금씩 다를 수 있다.

     > LLM 자체의 태생적 한계 → 사전에 output에 대한 강한 제한이 필요해 보인다.
- **요약**:
  → 동일 코드라도 LLM 응답이 랜덤하거나 문장형으로 출력되면 Enum 파서가 일치하지 않아 오류 발생.
  따라서 **출력 형식 명시 + 결정적 설정**이 중요.

---

08. 출력 수정 파서(OutputFixingParser)

> `OutputFixingParser`는 출력 파싱 과정에서 발생할 수 있는 오류를 자동으로 수정하는 기능을 제공한다. → 파싱의 안전장치. 한 파서가 제대로 작동하지 않으면 다른 파서를 써보거나, 형식 오류를 수정하는 방식으로 해결한다.
>
> ```
> OutputParserException: Invalid json output: {'name': 'Tom Hanks', 'film_names': ['Forrest Gump']}
> For troubleshooting, visit: https://python.langchain.com/docs/troubleshooting/errors/OUTPUT_PARSING_FAILURE
> ```
>
> 이런 식으로 사전에 설정한 스키마를 준수하지 않는 경우, 오류가 발생한다. 이런 오류를 자동으로 보완하기 위해 사용하는 것이다.

- 첫 번째 시도에서 **스키마를 준수하지 않는 결과**가 나올 경우, `OutputFixingParser` 가 자동으로 형식이 잘못된 출력을 인식하고, 이를 수정하기 위한 새로운 명령어와 함께 모델에 다시 제출한다.
  - 예시에 있던 형식이 잘못된 이유
    - `"{'name': 'Tom Hanks', 'film_names': ['Forrest Gump']}"` 의 경우, JSON 형식을 만족하지 못한다 (JSON에서는 `'` 대신 `"` 을 사용해야한다. `{"name": "Tom Hanks", "film_names": ["Forrest Gump"]}`
  - 이런 자잘한 형식 오류를 수정을 통해 JSON 형식에 맞게 자동으로 바꿔주는 것이 이번 OutputFixingParser의 역할이다.

    ```
    from langchain.output_parsers import OutputFixingParser

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

  - 잘 적용되어 에러 없이 출력된다.