

발표자료 2 (출력 파서)

- LangChain 에서 제공하는 주요 Output Parser 설명 및 사용 패턴 정리

1) 출력 파서

LLM의 출력 값을 더 유용하고 구조화된 형태로 해석해서 변환(파싱)하는 중요한 컴포넌트

- 스키마(schema) : 사전 정의된 양식

특징

- 다양성 : LangChain은 많은 종류의 출력 파서를 제공해 다양한 변환 방식에 대응
- 스트리밍 지원 : 실시간 데이터 처리 가능
- 확장성 : 확장 가능한 인터페이스를 제공해 최소한의 모듈에서 복잡한 모듈까지 필요한 기능 확장

장점

- 구조화 : LLM의 자유 형식 텍스트 출력을 구조화된 데이터로 변환해 정보를 체계적으로 관리
- 일관성 : 출력 형식을 일관되게 유지하여 후속 처리나 데이터 조치가 쉽고 효율적
- 유연성 : JSON, 리스트, 딕셔너리 등 다양한 출력 형식으로 변환 가능하여 유연성 제공

2) PydanticOutputParser

- pydantic : 파이썬에서 데이터 유효성을 검사하고, 데이터를 구조화된 형식으로 관리하는 데 유용한 라이브러리
- 유효성 검사(validation) : 코드를 실행할 때 데이터가 정해진 조건이나 형식에 맞는지 확인하여 잘못된 값이 입력되지 않도록 하는 도구

PydanticOutputParser

단순 텍스트 응답 대신 언어 모델의 출력을 특정 데이터 모델에 맞게 변환

- get_format_instructions() : 언어 모델이 출력해야 할 정보의 형식을 정의하는 지침 제공
ex) 언어 모델이 출력해야 할 데이터 필드와 그 형태를 설명하는 지침을 문자열로 반환
- parse() : 언어 모델의 출력(문자열로 가정)을 받아 이를 특정 구조로 분석하고 변환. Pydantic 과 같은 도구를 사용하여 입력된 문자열을 사전 정의된 스키마에 따라 검증하고, 해당 스키마를 따르는 데이터 구조로 변환

```
from pydantic import BaseModel, Field
```

```

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)

# instruction 출력
print(parser.get_format_instructions())

```

The output should be formatted as a JSON instance that conforms to the JSON schema below. As an example, for the schema {"properties": {"foo": {"title": "Foo", "description": "a list of strings", "type": "array", "items": {"type": "string"}}}, "required": ["foo"]} the object {"foo": ["bar", "baz"]} is a well-formatted instance of the schema. The object {"properties": {"foo": ["bar", "baz"]}} is not well-formatted. Here is the output schema: ``

```

{"properties": {"person": {"description": "메일을 보낸 사람", "title": "Person", "type": "string"}, "email": {"description": "메일을 보낸 사람의 이메일 주소", "title": "Email", "type": "string"}, "subject": {"description": "메일 제목", "title": "Subject", "type": "string"}, "summary": {"description": "메일 본문을 요약한 텍스트", "title": "Summary", "type": "string"}, "date": {"description": "메일 본문에 언급된 미팅 날짜와 시간", "title": "Date", "type": "string"}}, "required": ["person", "email", "subject", "summary", "date"]}

```

프롬프트 정의

1. question : 유저 질문 받기
2. email_conversation : 이메일 본문 내용 입력
3. format: 형식 지정

```

prompt = PromptTemplate.from_template(
    """
    You are a helpful assistant. Please answer the following questions in KOREAN.

    QUESTION:
    {question}

    EMAIL CONVERSATION:
    {email_conversation}

```

```

FORMAT:
{format}
"""
)

# format 에 PydanticOutputParser의 부분 포매팅(partial) 추가
prompt = prompt.partial(format=parser.get_format_instructions())

```

```

# 체인 생성
chain = prompt | llm

# chain 을 실행하고 결과를 출력합니다.
response = chain.stream(
    {
        "email_conversation": email_conversation,
        "question": "이메일 내용중 주요 내용을 추출해 주세요.",
    }
)

# 결과는 JSON 형태로 출력됩니다.
output = stream_response(response, return_output=True)

```

결과

```

{
  "person": "김철수",
  "email": "chulsoo.kim@bikecorporation.me",
  "subject": "\"ZENESIS\" 자전거 유통 협력 및 미팅 일정 제안",
  "summary": "바이크코퍼레이션 김철수 상무가 ZENESIS 자전거의 상세 브로슈어(기술 사양, 배터리 성능, 디자인) 요청과 유통 및 마케팅 협력 논의를 위해 1월 15일 오전 10시에 미팅을 제안함.",
  "date": "1월 15일 오전 10시"
} ``

```

output : 문자열 -> 객체 타입으로 변환

```

```python
structured_output = parser.parse(output)
print(structured_output)

```

```

EmailSummary(person='김철수', email='chulsoo.kim@bikecorporation.me',
subject='"ZENESIS" 자전거 유통 협력 및 미팅 일정 제안', summary='바이크코퍼레이션 김

```

철수 상무가 ZENESIS 자전거의 상세 브로슈어(기술 사양, 배터리 성능, 디자인) 요청과 유통 협력 및 마케팅 계획 구체화를 위해 미팅을 제안함.', date='1월 15일 오전 10시')

출력 값이 객체 타입이 되면 정형화된 데이터에서 키 값만 지정해서 원하는 정보를 쉽게 뽑아낼 수 있음

```
structured_output.person
```

```
`김철수`
```

체인을 생성해 결과를 출력하면 결과값 자체가 객체로 변환

```
chain = prompt | llm | parser

response = chain.invoke(
 {
 "email_conversation" : email_conversation,
 "question": "이메일 내용 중 주요 내용을 추출해 주세요"
 }
)

response
```

with\_structured\_output()

```
llm_with_structured = ChatOpenAI(
 temperature=0, model_name="gpt-4.1-mini"
).with_structured_output(EmailSummary)

invoke() 함수를 호출하여 결과 출력
answer = llm_with_structured.invoke(email_conversation)
```

.with\_structured\_output() 함수는 stream 기능을 지원하지 않음

### 3) LangSmith에서 출력파서 흐름 확인

- 입력값과 출력값이 잘 들어갔는지 확인

Waterfall

RunnableSequence

2.65s 733

PromptTemplate

0.00s seq:step:1

ChatOpenAI gpt-4.1-mini

2.65s 733 seq:step:2

PydanticOutputParser

0.00s seq:step:3

Run Feedback Metadata

검찰수  
상무이사  
바이크코퍼레이션

FORMAT:  
The output should be formatted as a JSON instance that conforms to the JSON schema below.  
  
As an example, for the schema {"properties": {"foo": {"title": "Foo", "description": "a list of strings", "type": "array", "items": {"type": "string"}}}, "required": ["foo"]}  
the object {"foo": ["bar", "baz"]} is a well-formatted instance of the schema. The object {"properties": {"foo": {"bar", "baz"}}} is not well-formatted.  
  
Here is the output schema:  
...  
{  
 "properties": {  
 "person": {"description": "메일을 보낸 사람", "title": "Person", "type": "string", "email": {"description": "메일을 보낸 사람의 이메일 주소", "title": "Email", "type": "string"}, "subject": {"description": "메일 제목", "title": "Subject", "type": "string"}, "summary": {"description": "메일 본문을 요약한 텍스트", "title": "Summary", "type": "string"}, "date": {"description": "메일 본문에 언급된 미팅 날짜와 시간", "title": "Date", "type": "string"}, "required": ["person", "email", "subject", "summary", "date"]  
 }  
}

Output

AI  
```json  
{
 "person": "김철수",
 "email": "chulsoo.kim@bikecorporation.me",
 "subject": "\"ZENESIS\" 자전거 유통 협력 및 미팅 일정 제안",
 "summary": "바이크코퍼레이션 김철수 상무가 ZENESIS 자전거의 상세 브로슈어(기술 사양, 배터리 성능, 디자인) 요청과 유통 협력 및 미팅을 제안함.",
 "date": "1월 15일 오전 10시"
}
...`

문자열 형식의 답변(Input)을 받아서 구조화된 객체(Rendered Output)로 변환

```
AI
```json
{
 "person": "김철수",
 "email": "chulsoo.kim@bikecorporation.me",
 "subject": "\"ZENESIS\" 자전거 유통 협력 및 미팅 일정 제안",
 "summary": "바이크코퍼레이션 김철수 상무가 ZENESIS 자전거의 상세 브로슈어(기술 사양, 배터리 성능, 디자인) 요청과 유통 협력 및 미팅을 제안함.",
 "date": "1월 15일 오전 10시"
}
```

## Output

### Output

- > **Date** 1월 15일 오전 10시
- > **Email** chulsoo.kim@bikecorporation.me
- > **Person** 김철수
- > **Subject** "ZENESIS" 자전거 유통 협력 및 미팅 일정 제안
- > **Summary** 바이크코퍼레이션 김철수 상무가 ZENESIS 자전거의 상세 브로슈어(기술 사양, 배터리 성능, 디자인) 요청과 유통...

## 4) 쉼표로 구분된 리스트 출력 파서

CommaSeparatedListOutputParser : 쉼표로 구분된 항목을 리스트 형식으로 처리하는 출력 파서

CSV (Comma Separated Values)

```
from langchain_core.output_parsers import CommaSeparatedListOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

콤마로 구분된 리스트 출력 파서 초기화
output_parser = CommaSeparatedListOutputParser()

출력 형식 지침 가져오기
format_instructions = output_parser.get_format_instructions()
프롬프트 템플릿 설정
prompt = PromptTemplate(
 # 주제에 대한 다섯 가지를 나열하라는 템플릿
 template="List five {subject}.\n{format_instructions}",
 input_variables=["subject"], # 입력 변수로 'subject' 사용
 # 부분 변수로 형식 지침 사용
 partial_variables={"format_instructions": format_instructions},
)

ChatOpenAI 모델 초기화
model = ChatOpenAI(temperature=0)

프롬프트, 모델, 출력 파서를 연결하여 체인 생성
chain = prompt | model | output_parser

"대한민국 관광명소"에 대한 체인 호출 실행
chain.invoke({"subject": "대한민국 관광명소"})

스트림을 순회합니다.
for s in chain.stream({"subject": "대한민국 관광명소"}):
 print(s) # 스트림의 내용을 출력합니다.
```

```
['경복궁'] ['남산타워'] ['부산 해운대해수욕장'] ['제주도'] ['경주여행']
```

리스트 형식으로 데이터를 받으면 파이썬의 리스트 문법을 활용해 데이터에 접근하거나 조작하기 편리

단, 출력 파서를 스트리밍 출력하면 토큰 단위로 별개의 리스트가 되어 구분되므로 활용성이 떨어짐

## 5) 구조화된 출력 파서

StructuredOutputParser

LLM에 대한 답변을 딕셔너리 형식으로 정의하고, 키와 값의 쌍으로 여러 필드를 변환하는 데이터 구조화에 유용

인텔리전스가 낮은 로컬 모델에 적합

ResponseSchema 클래스를 사용하여 사용자의 질문에 대한 답변과 사용된 소스(웹사이트)에 대한 설명을 포함하는 응답 스키마 정의

```
from langchain.output_parsers import ResponseSchema, StructuredOutputParser

사용자의 질문에 대한 답변
response_schemas = [
 ResponseSchema(name="answer", description="사용자의 질문에 대한 답변"),
 ResponseSchema(
 name="source",
 description="사용자의 질문에 답하기 위해 사용된 `출처`, `웹사이트주소` 이어야 합니다.",
),
]

응답 스키마를 기반으로 한 구조화된 출력 파서 초기화
output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
```

name : 응답 구조에서 각 항목의 키 이름 지정

answer : 질문에 대한 답변

description : 답변에 대한 설명.

source : 질문에 답하기 위해 사용된 출처와 웹사이트 주소를 담도록 정의. description에서 명시  
💡 로컬 모델에서는 인텔리전스가 부족할 수 있기 때문에 description의 내용을 영어로 작성하는 게 더 낫다

```
출력 형식 지시사항을 파싱합니다.
format_instructions = output_parser.get_format_instructions()
prompt = PromptTemplate(
 # 사용자의 질문에 최대한 답변하도록 템플릿을 설정합니다.
 template="answer the users question as best as possible.\n{format_instructions}\n{question}",
 # 입력 변수로 'question'을 사용합니다.
 input_variables=["question"],
 # 부분 변수로 'format_instructions'을 사용합니다.
 partial_variables={"format_instructions": format_instructions},
)
```

```

model = ChatOpenAI(temperature=0) # ChatOpenAI 모델 초기화
chain = prompt | model | output_parser # 프롬프트, 모델, 출력 파서를 연결

대한민국의 수도가 무엇인지 질문합니다.
chain.invoke({"question": "대한민국의 수도는 어디인가요?"})

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

 # 스트리밍 출력

 print(s)

```

```

{'answer': '서울', 'source':
 '[https://ko.wikipedia.org/wiki/%EC%84%9C%EC%9A%B8]
 (https://ko.wikipedia.org/wiki/%EC%84%9C%EC%9A%B8)'}

```

## 6) JSON 형식 출력 파서

JSONOutputParser

: LLM이 데이터를 조회하고 결과를 도출할 때 답변으로 단순한 문자열이 아니라 저장된 스키마에 맞게 JSON 형식으로 데이터를 반환.

LLM이 데이터를 정확하고 효율적으로 처리하여 사용자가 원하는 형태의 JSON을 생성하기 위해서는 모델의 용량이 충분히 커야 합니다.

ex) llama-8B 모델(매개변수 개수 80억)보다 더 큰 용량을 가지고 있어 보다 복잡한 데이터를 처리하는데 유리.

용량이 작은 모델에서는 JsonOutputParser를 쓸 때 오류 발생

```

from langchain_core.output_parsers import JsonOutputParser
from langchain_openai import ChatOpenAI
from pydantic import BaseModel, Field

OpenAI 객체를 생성합니다.
model = ChatOpenAI(temperature=0, model_name="gpt-4.1-mini")

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

```

```

질의 작성
question = "지구 온난화의 심각성 대해 알려주세요."

```



```

파서를 설정하고 프롬프트 템플릿에 지시사항을 주입합니다.
parser = JsonOutputParser(pydantic_object=Topic)
print(parser.get_format_instructions())

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

prompt = prompt.partial(format_instructions=parser.get_format_instructions())

체인을 구성합니다.
chain = prompt | model | parser

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

타입을 확인합니다.
type(answer) # dict

답변 확인
answer

```

```

{'description': '지구 온난화는 대기 중 온실가스 증가로 지구 평균 기온이 상승하는 현상으로, 극심한 기후 변화, 해수면 상승, 생태계 파괴 등 심각한 환경 문제를 초래합니다.',
'hashtags': '#지구온난화 #기후변화 #환경문제 #온실가스'}

```

## Pydantic 을 사용하지 않고 JSONOutputParser 사용

```

#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': '지구 온난화는 대기 중 온실가스 농도의 증가로 인해 지구 평균 기온이 상승하는 현상입니다. 이는 극지방의 빙하 감소, 해수면 상승, 기상이변 증가 등 다양한 환경 변화를 초래합니다.', 'hashtags': ['#지구온난화', '#기후변화', '#온실가스', '#환경보호', '#지구환경']}

```

## 7) Pandas 데이터 프레임 출력파서

Pandas 는 파이썬에서 엑셀이나 CSV 데이터를 불러와 정제, 변환, 분석 등 구조화된 데이터를 처리하는 도구. Pandas에서는 행과 열로 구성된 2차원 데이터 구조인 데이터프레임(DataFrame)으로 데이터 처리.

Pandas 데이터프레임 출력 파서는 임의의 Pandas 데이터프레임에서 데이터를 추출하고 이를 형식화된 딕셔너리 형태로 조회하며 연산 결과를 반환할 수도 있습니다.

데이터 구조를 보기 좋게 출력하게 위해 pprint 모듈.

변수, 함수 인자, 반환값 등의 데이터 타입을 명확하게 정의하는 typing 모듈

```

import pprint
from typing import Any, Dict

import pandas as pd
from langchain.output_parsers import PandasDataFrameOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

```

```

ChatOpenAI 모델 초기화 (gpt-3.5-turbo 모델 사용을 권장합니다)
model = ChatOpenAI(temperature=0, model_name="gpt-3.5-turbo")

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)
```

데이터를 읽어온 뒤 DataFrame을 로드하여 dict 변수에 할당  
PandasDataFrameOutputParser를 사용하여 DataFrame 파싱

```
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())
```

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

날짜 및 시간 출력 파서
output_parser = DatetimeOutputParser()
output_parser.format = "%Y-%m-%d"

사용자 질문에 대한 답변 템플릿
template = """Answer the users question:\n\n#Format Instructions:
\n{format_instructions}\n\n#Question: {question}\n\n#Answer: """

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

프롬프트 내용을 출력
prompt
```

프롬프트 템플릿에 지시사항 주입

행과 열을 조회하는 규칙 방식이나 데이터프레임의 열 이름을 알 수 있음

The output should be formatted as a string as the operation, followed by a colon, followed by the column or row to be queried on, followed by optional array parameters.

1. The column names are limited to the possible columns below.
2. Arrays must either be a comma-separated list of numbers formatted as [1,3,5], or it must be in range of numbers formatted as [0..4].
3. Remember that arrays are optional and not necessarily required.
4. If the column is not in the possible columns or the operation is not a valid Pandas DataFrame operation, return why it is invalid as a sentence starting with either "Invalid column" or "Invalid operation".

As an example, for the formats:

1. String "column:num\_legs" is a well-formatted instance which gets the column num\_legs, where num\_legs is a possible column.
2. String "row:1" is a well-formatted instance which gets row 1.
3. String "column:num\_legs[1,2]" is a well-formatted instance which gets the column num\_legs for rows 1 and 2, where num\_legs is a possible column.
4. String "row:1[num\_legs]" is a well-formatted instance which gets row 1, but for just column num\_legs, where num\_legs is a possible column.
5. String "mean:num\_legs[1..3]" is a well-formatted instance which takes the mean of num\_legs from rows 1 to 3, where num\_legs is a possible column and mean is a valid Pandas DataFrame operation.
6. String "do\_something:num\_legs" is a badly-formatted instance, where do\_something is not a valid Pandas DataFrame operation.
7. String "mean:invalid\_col" is a badly-formatted instance, where invalid\_col is not a possible column.

Here are the possible columns:

PassengerId, Survived, Pclass, Name, Sex, Age, SibSp, Parch, Ticket, Fare, Cabin, Embarked

ChatGPT의 말:

19s 동안 생각함

Got it. I'm ready to validate and format requests using your rules and the allowed columns.

Accepted operations (for clarity):

column, row

Aggregations on a single column: mean, sum, min, max, median, std, count, unique, nunique

Array syntax (optional): [1,3,5] or [0..4]

Allowed columns: PassengerId, Survived, Pclass, Name, Sex, Age, SibSp, Parch, Ticket, Fare, Cabin, Embarked

Send me a request in plain language (e.g., “average Fare for rows 1 to 10”), and I’ll return a single formatted string or an error starting with “Invalid column” / “Invalid operation”.

Quick examples I would return:

column:Age

row:1

column:Fare[1,2]

row:3[Name]

mean:Fare[1..3]

Invalid operation: do\_something is not a valid Pandas DataFrame operation

Invalid column: invalid\_col is not in the allowed columns

## 컬럼에 대한 값 조회

```
열 작업 예시입니다.
df_query = "Age column 을 조회해 주세요."

프롬프트 템플릿을 설정합니다.
prompt = PromptTemplate(
 template="Answer the user query.\n{format_instructions}\n{question}\n",
 input_variables=["question"], # 입력 변수 설정
 partial_variables={
 "format_instructions": parser.get_format_instructions()
 }, # 부분 변수 설정
)

체인 생성
chain = prompt | model | parser

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

# 출력

```
format_parser_output(parser_output)
```

```
{'Age': {0: 22.0, 1: 38.0, 2: 26.0, 3: 35.0, 4: 35.0, 5: nan, 6: 54.0, 7: 2.0,
8: 27.0, 9: 14.0, 10: 4.0, 11: 58.0, 12: 20.0, 13: 39.0, 14: 14.0, 15: 55.0,
16: 2.0, 17: nan, 18: 31.0, 19: nan}}
```

특정 열에서 일부 행의 평균을 검색하는 작업 예제입니다.

```
row 0 ~ 4의 평균 나이를 구합니다.
```

```
df["Age"].head().mean()
```

[13] ✓ 0.0s

... 31.2

```
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({"question": df_query})
```

```
결과 출력
```

```
print(parser_output)
```

[14] ✓ 0.5s

... {'mean': 31.2}

## 8) 날짜 형식 출력 파서

DateTimeOutputParser

파이썬에서 시간을 특정 형식으로 강제하여 표현할 때 사용되는 코드 형식

## 참고

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
%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



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

날짜 및 시간 출력 파서
output_parser = DatetimeOutputParser()
output_parser.format = "%Y-%m-%d"

사용자 질문에 대한 답변 템플릿
template = """Answer the users question:\n\n#Format Instructions:
\n{format_instructions}\n\n#Question: {question}\n\n#Answer: """

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

chain 생성
chain = prompt | ChatOpenAI() | output_parser

체인을 호출하여 질문에 대한 답변을 받습니다.
output = chain.invoke({"question": "Google 이 창업한 연도"})

결과를 문자열로 반환
output.strftime("%Y-%m-%d")
```

1998-09-04

## 9) 열거형 출력 파서

Enum(enumeration) : 동등한 레벨에 있는 데이터들을 하나의 구조로 묶어 표현하는 방식

EnumOutputParser : 언어 모델의 출력을 미리 정의된 열거형 값 중 하나로 파싱하는 도구.

LLM에게 정해진 선택지를 미리 전달하고 그중에서 하나를 골라서 답변할 수 있게 하므로 출력 데이터의 일관성을 유지하고 예측 가능성을 높일 수 있습니다.

```
from langchain.output_parsers.enum import EnumOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI
```

```

from enum import Enum

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

EnumOutputParser 인스턴스 생성
parser = EnumOutputParser(enum=Colors)

프롬프트 템플릿을 생성합니다.
prompt = PromptTemplate.from_template(
 """다음의 물체는 어떤 색깔인가요?
 Object: {object}
 Instructions: {instructions}"""
 # 파서에서 지시사항 형식을 가져와 부분적으로 적용합니다.
).partial(instructions=parser.get_format_instructions())

체인을 생성합니다.
chain = prompt | ChatOpenAI() | parser

체인을 실행합니다.
response = chain.invoke({"object": "하늘"})

```

```
Colors.BLUE
```

```

출력의 타입을 확인합니다.
type(response)

```

```
<enum 'Colors'>
```

```

출력의 값 확인
response.value

```

```
"파란색"
```

## 10 )OutputFixingParser

출력 파싱 과정에서 발생할 수 있는 오류를 자동으로 수정하는 기능 제공.

기본적으로 다른 파서(ex. PydanticOutputParser)를 래핑하고, 이 파서가 처리할 수 없는 형식의 출력이나 오류를 반환할 경우, 추가적인 LLM 호출을 통해 오류를 수정하도록 설계

첫 번째 시도에서 스키마를 준수하지 않는 결과가 나올 경우, OutputFixingParser 가 자동으로 형식이 잘못된 출력을 인식하고, 이를 수정하기 위한 새로운 명령어와 함께 모델에 다시 제출하는 것.

수정을 위한 명령어는 오류를 정확히 지적하고, 올바른 형식으로 데이터를 재구성할 수 있도록 구체적인 지시를 포함해야 함.

```
from langchain_openai import ChatOpenAI
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field
from typing import List

class Actor(BaseModel):
 name: str = Field(description="name of an actor")
 film_names: List[str] = Field(description="list of names of films they starred in")

actor_query = "Generate the filmography for a random actor."

parser = PydanticOutputParser(pydantic_object=Actor)
```

## 오류 발생

```
잘못된 형식을 일부러 입력
misformatted = '{"name': 'Tom Hanks', 'film_names': ['Forrest Gump']}'

잘못된 형식으로 입력된 데이터를 파싱하려고 시도
parser.parse(misformatted)

오류 출력
```

```
OutputParserException: Invalid json output: {'name': 'Tom Hanks',
'film_names': ['Forrest Gump']}
```

```
from langchain.output_parsers import OutputFixingParser

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

```
OutputFixingParser 를 사용하여 잘못된 형식의 출력을 파싱
actor = new_parser.parse(misformatted)
```