출력 파서 (Output Parser)

1.1 출력파서 개요

1.1 출력파서 개요

LangChain의 **출력 파서(Output Parser)** 는 LLM이 생성한 텍스트 결과를 원하는 형식으로 **구조화**하거나 **후처리(post-processing)** 하는 도구이다.
즉, 모델이 "자연어 문장"을 출력하더라도, 이를 파서를 통해 **숫자, JSON, 선택지, 불리언, 리스트** 같은 명확한 형식으로 변환할 수 있다.

---

1. 출력 파서의 필요성

LLM은 확률적으로 텍스트를 생성하기 때문에:

- **정확한 JSON 형식**을 요구해도 누락, 오타, 불필요한 설명이 포함될 수 있음.
- **숫자/객관식 선택지**가 필요한 경우 불필요한 단어가 붙을 수 있음.
  → 이런 문제를 해결하기 위해 출력 파서를 사용한다.

1.1 출력파서 개요

2. 주요 출력 파서 종류

출력 파서	설명	사용 예시
StrOutputParser	단순히 문자열 그대로 반환	자유 텍스트 요약, 챗봇 답변
RegexParser	정규표현식으로 결과 추출	`"정답: 42"` → `42`
PydanticOutputParser	[pydantic] 모델을 활용해 JSON → Python 객체 변환	구조화된 JSON 응답
StructuredOutputParser	사전(schema) 기반으로 JSON 파싱	`{ "name": ..., "age": ... }`
DatetimeOutputParser	날짜·시간 형식으로 변환	`"2025-09-30"`
CommaSeparatedListOutputParser	문자열을 리스트로 변환	`"사과, 배, 포도"` → `["사과","배","포도"]`
BooleanOutputParser	True/False 값으로 변환	`"예"` → `True`
RetryOutputParser	실패 시 재시도 수행	JSON 형식 깨짐 방지

1.1 출력파서 개요

3. 실제 활용 예시

- **객관식 문제 풀이 자동화**: "정답: (a)" 형식에서 알파벳만 추출
- **RAG 파이프라인**: 문서 검색 후 답변을 JSON `{ "answer": "...", "references": [...] }` 형식으로 변환
- **데이터 전처리**: LLM 출력값을 리스트, 숫자 등으로 자동 변환해 후속 처리에 활용

👉 정리하면, **LangChain 출력 파서 = LLM의 자유로운 텍스트 출력을 신뢰할 수 있는 구조화된 데이터로 바꿔주는 모듈**이다.

1.2 StrOutputParser

1.2 StrOutputParser

StrOutputParser 는 LangChain에서 제공하는 출력 파서 중 가장 기본적인 형태다.

---

1. 정의

- LLM이 생성한 **출력 텍스트 그대로 반환**하는 파서이다.
- 별도의 구조화, 변환, 검증 같은 기능은 수행하지 않는다.
- 따라서 **자유 형식 답변(문장, 요약, 설명 등)** 을 받을 때 적합하다.

---

2. 예제 코드

```python
from langchain.output_parsers import StrOutputParser

parser = StrOutputParser()

result = parser.parse("안녕하세요! LangChain 출력 파서 예제입니다.")
print(result)
# 출력: "안녕하세요! LangChain 출력 파서 예제입니다."
```

1.2 StrOutputParser

3. 특징

- **가볍고 단순** → 오버헤드 거의 없음
- **후처리 필요 없는 경우** 사용
- 다른 파서(Pydantic, Regex 등)와 달리 실패하거나 에러가 발생할 일이 거의 없음

---

4. 사용 사례

- 일반 챗봇 답변
- 요약 결과 텍스트 반환
- 번역 결과 그대로 출력
- 자유로운 창작 텍스트 생성

👉 정리:

StrOutputParser = LLM 출력 텍스트를 그대로 반환하는 가장 단순한 파서이다.

1.3 PydanticOutputParser

1.3 PydanticOutputParser

1. Pydantic이란?

Pydantic은 Python에서 데이터 검증과 설정 관리를 쉽게 할 수 있도록 도와주는 **데이터 검증 라이브러리**이다.

- Python의 **타입 힌트(type hint)** 를 기반으로 동작한다.
- 입력 데이터를 검사하고, 올바른 타입으로 자동 변환해 준다.
- 잘못된 데이터가 들어오면 **ValidationError** 를 발생시켜 안전성을 높인다.

👉 한마디로 **"데이터를 정해진 스키마(schema)대로 안전하게 다루게 해주는 도구"**이다.

1.3 PydanticOutputParser

2. 간단한 예제

```python
from pydantic import BaseModel

# 데이터 스키마 정의
class User(BaseModel):
    id: int
    name: str
    age: int

# 올바른 데이터 → 정상 객체 생성
user = User(id=1, name="홍길동", age=25)
print(user)
# User(id=1, name='홍길동', age=25)

# 잘못된 데이터 → 자동 변환 또는 에러
user2 = User(id="2", name="이몽룡", age="30")
print(user2)
# User(id=2, name='이몽룡', age=30)   ← 문자열이 int로 자동 변환됨

# 타입 불일치 → 에러 발생
user3 = User(id="삼", name="성춘향", age="스무살")
# pydantic.error_wrappers.ValidationError 발생
```

1.3 PydanticOutputParser

3. 주요 기능

1. **데이터 검증 (Validation)**
   - 타입 자동 변환 (문자열 `"123"` → 정수 `123`)
   - 불가능할 경우 오류 발생
2. **데이터 직렬화/역직렬화**
   - `.dict()` → Python dict
   - `.json()` → JSON 문자열
3. **모델 중첩(Nested Model)**
   - 객체 안에 또 다른 객체 포함 가능
4. **환경 변수 / 설정 관리**
   - `.env` 파일을 자동으로 읽어 설정값 불러오기 가능

1.3 PydanticOutputParser

4. 어디에 쓰이나?

- **FastAPI** (백엔드 프레임워크): 요청(request) 데이터 자동 검증
- **LangChain**: LLM 출력값(JSON)을 안전하게 Python 객체로 변환
- **데이터 처리/ETL**: 외부 API, CSV, DB에서 불러온 데이터 검증

✅ 정리

Pydantic은 **"Python 데이터 검증 및 직렬화 라이브러리"**이다.

타입 안정성을 확보하고, 잘못된 데이터가 들어왔을 때 안전하게 처리할 수 있도록 도와준다.

1.3 PydanticOutputParser

LangChain의 **PydanticOutputParser**는 LLM이 출력한 텍스트를 **Pydantic 모델**로 변환해 주는 출력 파서이다.

---

1. 특징

- LLM이 만든 JSON 같은 문자열을 자동으로 **Python 객체**로 바꿔준다.
- **유효성 검사(validation)** 기능이 내장되어 있어, 누락된 필드나 잘못된 데이터 타입을 잡아낼 수 있다.
- LLM이 JSON 형식을 틀리게 출력했을 때도, 자동으로 재시도하거나 오류를 알려줄 수 있다.

1.3 PydanticOutputParser

2. 기본 구조

```python
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field

# ① Pydantic 모델 정의
class Person(BaseModel):
    name: str = Field(description="이름")
    age: int = Field(description="나이")

# ② PydanticOutputParser 생성
parser = PydanticOutputParser(pydantic_object=Person)

# ③ 모델이 출력할 JSON 예시
text = '{"name": "홍길동", "age": 25}'

# ④ 파싱 실행
result = parser.parse(text)
print(result)        # Person(name='홍길동', age=25)
print(result.name)   # "홍길동"
print(result.age)    # 25
```

1.3 PydanticOutputParser

3. 장점

1. **자동 JSON 파싱**: 문자열 → Python 객체로 즉시 변환
2. **타입 보장**: `int`, `str`, `list` 등 자료형을 강제
3. **에러 방지**: LLM이 엉뚱한 값을 주면 에러 발생 → 디버깅 쉬움
4. **프롬프트 연동**: `.get_format_instructions()` 메서드로 LLM에게 출력 포맷 예시를 제공 가능

코드 복사

```
print(parser.get_format_instructions())
```

👉 이 코드를 실행하면 **"LLM은 반드시 { "name": ..., "age": ... } JSON을 출력하라"** 같은 지침을 자동 생성해 준다.

1.3 PydanticOutputParser

4. 활용 사례

- **질문/답변 파이프라인**: `"답변과 신뢰도 점수"`를 JSON으로 받아 구조화
- **RAG 검색**: `"answer, references"` 필드를 갖춘 응답 JSON 생성
- **데이터 수집**: 뉴스 기사에서 `"title, date, keywords"` 추출

✅ 정리:

Pydantic 출력 파서는 LLM의 출력을 **"스키마 기반 JSON → Python 객체"**로 변환하고 검증해주는 강력한 도구이다.

즉, 단순 텍스트가 아니라 **신뢰성 있는 구조화 데이터**가 필요할 때 필수적으로 사용된다.

1.4 CSV Parser :

CommaSeparatedListOutputParser

1.4 CSV Parser

👍 **CommaSeparatedListOutputParser**는 LLM이 출력한 문자열을 **쉼표(,) 기준으로 나눠 리스트**로 변환해 주는 파서이다.

특징

입력: "사과, 배, 포도"

출력: ["사과", "배", "포도"] (자동으로 리스트 변환)

데이터 전처리나 **목록형 응답**을 받을 때 유용하다.

1.4 CSV Parser

간단 예제

```
from langchain.output_parsers import CommaSeparatedListOutputParser

# ① 파서 생성
parser = CommaSeparatedListOutputParser()

# ② LLM 출력처럼 가정한 문자열
text = "사과, 배, 포도, 바나나"

# ③ 파싱 실행
result = parser.parse(text)

print(result)
# 출력: ['사과', '배', '포도', '바나나']
```

1.5 JSON Parser :

JsonOutputParser

1.5 JSON Parser

JsonOutputParser는 LangChain에서 LLM이 생성한 결과를 **JSON 형식**으로 안전하게 파싱해주는 출력 파서이다.

````
LLM 출력: ```json
{
  "country": "대한민국",
  "capital": "서울",
  "population": 51780579,
  "languages": ["한국어"],
  "major_cities": ["부산", "인천", "대구", "대전", "광주"]
}
```

파싱 결과: {'country': '대한민국', 'capital': '서울', 'population': 51780579,
'languages': ['한국어'], 'major_cities': ['부산', '인천', '대구', '대전', '광
주']}
````
````

1.5 JSON Parser

예제 코드

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

# 1) 파서 준비
parser = JsonOutputParser()

# 2) 프롬프트 정의 (한글 질의문)
prompt = PromptTemplate(
    template="""
다음 질문에 대한 답변을 JSON 형식으로 작성해 주세요.

질문: {question}

{format_instructions}
""",
    input_variables=["question"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)
```

1.5 JSON Parser

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

# 4) 실행
_input = prompt.format(question="대한민국의 수도와 인구는 무엇인가요?")
output = model.invoke(_input)

# 5) 파싱
result = parser.parse(output.content)

print("LLM 출력:", output.content)
print("파싱 결과:", result)
```

1.5 JSON Parser

동작 흐름

1. `get_format_instructions()`가 LLM에게 **JSON 형식 안내문**을 자동 제공
2. LLM은 `{ "capital": "...", "population": ... }` 같은 JSON 형식으로 답변
3. `JsonOutputParser`가 문자열(JSON) → Python dict로 변환

예상 결과 예시

```
LLM 출력: {"capital": "서울", "population": 51000000}
파싱 결과: {'capital': '서울', 'population': 51000000}
```

👉 정리:

JsonOutputParser = LLM이 출력한 JSON 문자열을 Python dict로 안전하게 변환하는 파서이다.

1.6 StructuredOutputParser

1.6 StructuredOutputParser

StructuredOutputParser 개요

- LangChain에서 **사전(schema) 기반 JSON 출력**을 강제할 수 있는 파서
- `JsonOutputParser` + `PydanticOutputParser` 와 달리, **간단한 필드 스키마**만 지정하고 싶은 경우 유용

예제 코드

```
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StructuredOutputParser, ResponseSchema
from langchain_openai import ChatOpenAI


# 1) 출력 스키마 정의
response_schemas = [
    ResponseSchema(name="capital", description="국가의 수도"),
    ResponseSchema(name="population", description="국가의 인구 (정수)"),
    ResponseSchema(name="languages", description="국가에서 사용되는 주요 언어, 쉼표로 구분")
]
```

1.6 StructuredOutputParser

```
# 2) 파서 생성
parser = StructuredOutputParser.from_response_schemas(response_schemas)

# 3) 프롬프트 템플릿 정의
prompt = PromptTemplate(
    template="""
다음 질문에 대해 반드시 JSON 형식으로 답변해 주세요.

질문: {question}

{format_instructions}
""",
    input_variables=["question"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)
```

1.6 StructuredOutputParser

```python
# 4) LLM 초기화
model = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# 5) 실행
_input = prompt.format(question="대한민국의 수도, 인구, 사용 언어를 알려주세요.")
output = model.invoke(_input)

# 6) 파싱
result = parser.parse(output.content)

print("LLM 출력:", output.content)
print("파싱 결과:", result)
```

1.6 StructuredOutputParser

예상 결과 예시

```
LLM 출력: {
  "capital": "서울",
  "population": 51700000,
  "languages": "한국어"
}

파싱 결과: {
  'capital': '서울',
  'population': '51700000',
  'languages': '한국어'
}
```

1.6 StructuredOutputParser

- 작은 모델은 자주 이런 출력 오류를 냄:
  - `"capital": "서울", "population": 오백만명` ← 숫자가 아니라 문자열/한글로 적음
  - `{ "capital": "서울", "population": }` ← 값 누락
  - `{ 수도: "서울" }` ← 키 이름 잘못 사용
- **Pydantic/JSON 파서**를 쓰면:
  - 올바른 JSON 스키마로 재시도 유도
  - 타입 검증 실패 → 에러 감지 후 fallback 처리
  - 결국 "불안정한 모델"도 **안전한 구조화 데이터**를 뽑아낼 수 있음

✅ 정리

- **StructuredOutputParser**는 단순한 JSON 스키마를 정의하고, LLM 출력이 그 구조를 따르도록 유도하는 파서
- **복잡한 모델 검증**이 필요하면 `PydanticOutputParser`를,
- **간단한 JSON 구조**만 필요하면 `StructuredOutputParser`를 쓰면 된다.

1.6 StructuredOutputParser

요약 비교

파서	특징	적합한 경우
StrOutputParser	그냥 문자열 반환	자유 텍스트
JsonOutputParser	일반 JSON → dict	기본 JSON 파싱
PydanticOutputParser	JSON + 타입/스키마 검증	복잡·엄격한 구조 필요
StructuredOutputParser	간단한 키-값 JSON	가벼운 JSON 구조, 빠른 프로토타이핑

✅ 정리

StructuredOutputParser는 "엄격한 검증은 필요 없고, 간단한 JSON 키-값만 빠르게 받고 싶을 때" 가장 적합합니다.

1.7 PandasDataFrameOutputParser

1.7 PandasDataFrameOutputParser

PandasDataFrameOutputParser 는 LLM의 출력 결과를 **판다스 DataFrame**으로 변환할 수 있게 해준다.

```
LLM 출력:
[
  {"city": "서울", "population": 9500000},
  {"city": "부산", "population": 3400000},
  {"city": "인천", "population": 2900000},
  {"city": "대구", "population": 2400000}
]

변환된 Pandas DataFrame:
   city   population
0   서울      9500000
1   부산      3400000
2   인천      2900000
3   대구      2400000
```

1.7 PandasDataFrameOutputParser

```
import json
import pandas as pd
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

prompt = PromptTemplate(
    template="""
다음 질문에 대해 반드시 JSON 배열 형식으로만 답하세요.
⚠️ 코드 블록(```) 없이 순수 JSON만 출력하세요.

질문: {question}
""",
    input_variables=["question"],
)

model = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

1.7 PandasDataFrameOutputParser

```
_input = prompt.format(question="대한민국의 주요 도시와 인구를 알려주세요.")
output = model.invoke(_input)

# JSON → DataFrame
data = json.loads(output.content)
df = pd.DataFrame(data)

print("LLM 출력:", output.content)
print("\n변환된 Pandas DataFrame:")
print(df)
```

	도시	인구
0	서울	9746000
1	부산	3406000
2	인천	2953000
3	대구	2426000
4	대전	1493000
5	광주	1503000
6	울산	1153000
7	세종	350000

1.8 DatetimeOutputParser

1.8 DatetimeOutputParser

이 파서는 LLM이 출력한 텍스트를 **Python datetime 객체**로 변환해 주는 역할을 한다.

1.8 DatetimeOutputParser

📌 기본 예제

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

# 1) 출력 파서 준비
parser = DatetimeOutputParser()

# 2) 프롬프트 정의
prompt = PromptTemplate(
    template="""
다음 질문에 대해 날짜/시간만 출력하세요.
⚠️ 반드시 {format_instructions} 형식으로 답변하세요.

질문: {question}
""",
    input_variables=["question"],
    partial_variables={"format_instructions": parser.get_format_instructions()},
)
```

1.8 DatetimeOutputParser

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

# 4) 실행
_input = prompt.format(question="대한민국이 건국된 날짜는 언제인가요?")
output = model.invoke(_input)

# 5) 파싱 → datetime 변환
dt = parser.parse(output.content)

print("LLM 출력:", output.content)
print("변환된 datetime 객체:", dt)
print("타입:", type(dt))
```

1.8 DatetimeOutputParser

📌 예상 결과

```
LLM 출력: 1948-08-15
변환된 datetime 객체: 1948-08-15 00:00:00
타입: <class 'datetime.datetime'>
```

✅ 활용 포인트

- 자유 텍스트로 "1948년 8월 15일" 같은 값이 와도 파서가 `datetime`으로 변환
- **날짜 연산, 정렬, 비교** 등 Python datetime 기능을 바로 활용 가능
- 형식 강제: `get_format_instructions()`가 자동으로 `"YYYY-MM-DD"` 같은 규칙을 모델에 안내

1.9 RegexParser

1.9 RegexParser

🔹 RegexParser란?

LangChain의 **출력 파서(Output Parser)** 중 하나로,
LLM이 출력한 텍스트에서 **정규표현식(Regular Expression)** 을 이용해
원하는 부분만 추출할 때 사용하는 파서입니다.

🔹 특징

- 문자열 안에서 **패턴 매칭**으로 필요한 값만 뽑음
- 예를 들어 `"정답: (c)"` → `'c'` 만 추출 가능
- 숫자, 날짜, 선택지, 특정 단어 등 정해진 형식의 데이터를 쉽게 파싱

1.9 RegexParser

- **사용 예시**

```
from langchain.output_parsers import RegexParser

parser = RegexParser(
    regex=r"정답:\s*\(([a-d])\)",
    output_keys=["answer"]
)
result = parser.parse("정답: (c)")
print(result)  # {'answer': 'c'}
```

1.9 RegexParser

🔹 정규표현식(Regular Expression, regex) 이란?

문자열 안에서 **특정 패턴을 찾거나**, **치환하거나**, **검증**하기 위한 "검색 언어"예요.
쉽게 말해, **문자열 패턴 매칭 도구**입니다.

예를 들어,

- 전화번호 찾기: `010-1234-5678`
- 이메일 찾기: `user@gmail.com`
- 숫자만 추출하기: `12345`

이런 걸 자동으로 찾아낼 수 있습니다.

1.9 RegexParser

🔹 기본 문법 요약

패턴	의미	예시
`.`	임의의 문자 1개	`a.b` → `acb`, `a1b` 등
`\d`	숫자	`\d\d\d` → 3자리 숫자
`\w`	영문자, 숫자, `_`	`\w+` → 단어 하나
`\s`	공백 문자(띄어쓰기, 탭 등)	`"a\sb"` → `"a b"`
`+`	1개 이상 반복	`\d+` → "숫자가 하나 이상"
`*`	0개 이상 반복	`\d*` → "숫자가 없거나 여러 개"
`?`	0개 또는 1개	`colou?r` → `color`, `colour`
`{n}`	n개 반복	`\d{4}` → 4자리 숫자
`[]`	문자 집합	`[abc]` → `a` 또는 `b` 또는 `c`
`^`	문자열의 시작	`^Hello` → "Hello"로 시작
`$`	문자열의 끝	`end$` → "end"로 끝
`()`	그룹화 / 추출	`(abc)` → 그룹 캡처

1.9 RegexParser

파이썬에서 사용 방법

```
import re

text = "전화번호는 010-1234-5678 입니다."
result = re.findall(r"\d{3}-\d{4}-\d{4}", text)
print(result)  # ['010-1234-5678']
```

감사합니다