Output Parsers

> **Output parsers** in LangChain are used to **convert raw output (text)** from the LLM into **structured data** (like Python dicts, objects, lists, etc).

- Think of them as the **translator** between what the LLM *says* (text) and what your *program* needs (structured data).

🛠️ Common Output Parsers in LangChain

Parser	Description	Example Output
`StrOutputParser()`	Default → **Just plain string**	`"Hello world"`
`JsonOutputParser()`	Converts **JSON** text → **Python dict**	`{"name": "John"}`
`PydanticOutputParser()`	Converts **JSON** text → **Pydantic object**	`User(name="John")`
`CommaSeparatedListOutputParser()`	Parses comma-separated values → **List**	`["apple", "banana"]`
`RetryWithErrorOutputParser()`	Tries again if parsing fails	(advanced)
`OutputFixingParser()`	Fixes malformed output	(advanced)

🎯 When to Use Which Parser?

Goal	Use This Parser
You just want plain text	`StrOutputParser()`
LLM returns JSON	`JsonOutputParser()`
You want strict structure + validation	`PydanticOutputParser()`
You need a list from comma-separated values	`CommaSeparatedListOutputParser()`
You want auto-fix of malformed outputs	`OutputFixingParser()`

Code example:

```
from langchain_core.output_parsers import StrOutputParser
from langchain.prompts import ChatPromptTemplate

prompt= ChatPromptTemplate(
    [("system", "you are a helpful assigtant."),
    ("human", "tell me joke about {input}")
      ]                    )

chain = prompt | llm | StrOutputParser()
response = chain.invoke({"input": "cat"})  # Raw string

response
```

```
"Why don't cats play poker in the jungle? \n\nToo many cheetahs! 😹 \n"
```

Most Common output parsers:

1. `StrOutputParser()` → `StrOutputParser()` (same)
2. `JsonOutputParser()` → `JsonOutputParser()` (same)
3. `StructuredOutputParser()` → **`StructuredOutputParser.from_response_schemas(schema)`**
4. `PydanticOutputParser()` → **`PydanticOutputParser(pydantic_object=Person)`**

1. `StrOutputParser()`

- We don't need ~~`result.content`~~ if we use **`StrOutputParser()`**

Use case:


```
from langchain_groq import ChatGroq
from dotenv import load_dotenv

from langchain_core.output_parsers import StrOutputParser
from langchain.prompts import PromptTemplate

load_dotenv()

model = ChatGroq(model="gemma2-9b-it")
```

- Now, define 2 Prompt Templates:

```
# 1st prompt → detailed report
template1 = PromptTemplate(
    template='Write a detailed report on {topic}',
    input_variables=['topic']
)

# 2nd prompt → summary
template2 = PromptTemplate(
    template='Write a 5 line summary on the following text. /n {text}',
    input_variables=['text']
)
```

Invoke individual prompt:

```
prompt1= template1.invoke({'topic': 'blackhole'})
prompt1
```

```
StringPromptValue(text='Write a detailed report on blackhole')
```

```
result= model.invoke(prompt1)
result
```

```
AIMessage(content="##  Black Holes: Cosmic Abyss\n\n**Abstract:** Black
```

- The result is not a plain string

Invoke template2:

```
prompt2 = template2.invoke({'text': result.content})


result = model.invoke(prompt2)
result.content
```

```
'Black holes are incredibly dense objects formed from the collapse of
```

Same code with `StrOutputParser()`

```
parser = StrOutputParser()
```

```
chain = template1 | model | parser | template2 | model | parser
```

```
result= chain.invoke({'topic': 'blackhole'})
result
```

```
'Black holes are regions of spacetime with immense gravit
```

Full code:

```
# Full code

from langchain_groq import ChatGroq
from dotenv import load_dotenv

from langchain_core.output_parsers import StrOutputParser
from langchain.prompts import PromptTemplate

load_dotenv()

model = ChatGroq(model="gemma2-9b-it")

# 1st prompt → detailed report
template1 = PromptTemplate(
    template='Write a detailed report on {topic}',
    input_variables=['topic']
)

# 2nd prompt → summary
template2 = PromptTemplate(
    template='Write a 5 line summary on the following text. /n {text}',
```

```
    input_variables=['text']
)


parser= StrOutputParser()

chain = template1 | model | parser | template2 | model | parser

chain.invoke("potato")
```

```
'The potato, originating in the Andes over 9,000 years ago, has become
```

💡 You can also write `{'topic':'potato'}`

2. `JsonOutputParser()`

- Forces an LLM to give output in JSON format.

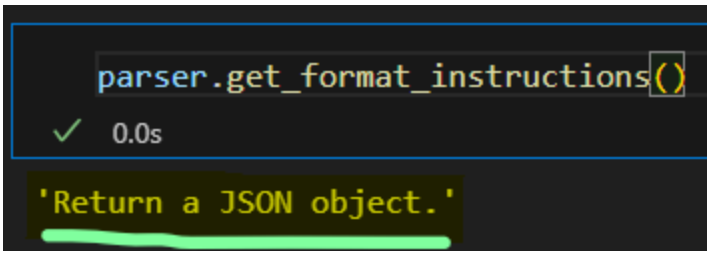
```
from langchain_groq import ChatGroq
from dotenv import load_dotenv

from langchain_core.output_parsers import JsonOutputParser
from langchain.prompts import PromptTemplate

load_dotenv()

model = ChatGroq(model="gemma2-9b-it")
```

Define a parer:

```
parser = JsonOutputParser()
```

```
template = PromptTemplate(
    template='Give me 5 facts about {topic} \n {format_instruction}',
    input_variables=['topic'],
    partial_variables={'format_instruction': parser.get_format_instructions()}
)
```

- `{format_instruction}` → You tell the LLM what kind of output you want
  - Parser tells you this instruction when you call `.get_format_instructions()` function
  - It prints → ***Return a JSON object***



- `partial_variables` gets filled before runtime

```
chain = template | model | parser

result = chain.invoke({"topic": "black hole"})

result
```

```
{'facts': ['Black holes are regions in spacetime where gravity is so strong that nothing, not even light, can escape.',
  'They are formed when massive stars collapse at the end of their life cycle.',
  'Black holes are characterized by their event horizon, a boundary beyond which escape is impossible.',
  'Although we cannot see black holes directly, their presence can be inferred by observing their effects on surrounding matter.',
  'The most common type of black hole is a stellar-mass black hole, which is a few times more massive than our Sun.']}
```

💡 **Major flaw in `JsonOutputParser()` → You cannot define a schema.❌**

- `StructuredOutputParser()` solves this problem.

3. `StructuredOutputParser()`

- `StructuredOutputParser` is used to **convert the raw text output** from a language model into **structured Python data** — like a dictionary, list, or custom object — using format instructions you define.

This helps when:

- You want **data validation**
- You need **structured formats** (JSON, list, object)
- You want to **auto-correct** or retry if structure breaks

> You provide a **schema**

```
from langchain_groq import ChatGroq
from dotenv import load_dotenv
from langchain.output_parsers import StructuredOutputParser, ResponseSche
ma
from langchain.prompts import PromptTemplate

load_dotenv()

model = ChatGroq(model="gemma2-9b-it")
```

> `StructuredOutputParser` is in `langchain.output_parsers`, not in ~~`langchain_core.output_parsers`~~ unlike others

- First, you define a schema with the help of `ResponseSchema`
- **The structure looks like:**

```
schema = [
ResponseSchema (),
ResponseSchema (),
ResponseSchema ()
]
```

```
schema = [
    ResponseSchema(name='fact_1', description='Fact 1 about the topic'),
    ResponseSchema(name='fact_2', description='Fact 2 about the topic'),
    ResponseSchema(name='fact_3', description='Fact 3 about the topic')

]
```

Create a parser object:

```
parser = StructuredOutputParser.from_response_schemas(schema)
```

Create a prompt:

```
template = PromptTemplate(
    template= "Give 3 facts about {topic} \n {format_instruction}",
    input_variables= ['topic'],
    partial_variables = {'format_instruction': parser.get_format_instructions()}
)
```

- Exactly same as the previous prompt

Invoke:

```
chain = template | model | parser

result = chain.invoke({'topic':'langchain'})
result
```

```
{'fact_1': 'LangChain is an open-source framework designed to simplify the development of applications powered by large language models (LLMs).',
 'fact_2': 'It provides tools for tasks like prompt engineering, chain creation, memory management, and integration with various LLMs and data sources.',
 'fact_3': 'LangChain enables developers to build more sophisticated and customizable AI applications by chaining together different LLM components and external s
```

💡 **Disadvantage of `StructuredOutputParser` → No validation❌**

- `PydanticOutputParser()` solves this issue.

4. `PydanticOutputParser()`

- `PydanticOutputParser` is used to **parse structured LLM output into Python objects** using Pydantic models — which are very powerful tools for validating and structuring data.

It's like saying:

> "Hey LLM, please give me the answer in this strict format... and if it messes up, we'll catch or fix it."

🔍 Why Should You Use It?

Use `PydanticOutputParser` if you want:

- **Strict JSON structure**
- **Typed fields (int, str, list, etc)**
- **Automatic error handling & validation**

- **Cleaner code & reliable outputs**

```
from langchain_groq import ChatGroq
from dotenv import load_dotenv

from langchain_core.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field

from langchain.prompts import PromptTemplate

load_dotenv()

model = ChatGroq(model="gemma2-9b-it")
```

Create a Pydantic object:

```
class Person(BaseModel):
    name: str = Field(description="Name of the person")
    age: int = Field(gt=18, description="age of the person")
    city : str =  Field(description='Name of the city the person belongs to')
```

Create a parser:

```
parser = PydanticOutputParser(pydantic_object=Person)
```

Write a prompt:

```
template = PromptTemplate(
    template='Generate the name, age and city of a fictional {place} person \n
{format_instruction}',
    input_variables=['place'],
```

```
    partial_variables={'format_instruction': parser.get_format_instructions()}
)
```

FYI:

```
parser.get_format_instructions()
```

```
Output:
'The output should be formatted as a JSON instance that conforms to the J
SON schema below.\n\nAs an example, for the schema {"properties": {"fo
o": {"title": "Foo", "description": "a list of strings", "type": "array", "items":
{"type": "string"}}}, "required": ["foo"]}\nthe object {"foo": ["bar", "baz"]} i
s a well-formatted instance of the schema. The object {"properties": {"fo
o": ["bar", "baz"]}} is not well-formatted.\n\nHere is the output schema:\n``
`\n{"properties": {"name": {"description": "Name of the person", "title": "N
ame", "type": "string"}, "age": {"description": "age of the person", "exclusi
veMinimum": 18, "title": "Age", "type": "integer"}, "city": {"description": "Na
me of the city the person belongs to", "title": "City", "type": "string"}}, "req
uired": ["name", "age", "city"]}\n```':
```

```
chain = template | model | parser

final_result = chain.invoke({'place':'indian'})

print(final_result)
```

```
name='Anika Sharma' age=25 city='Mumbai'
```