LangChain: Output Parsers

In this notebook we learn **Output Parsers** in LangChain v1.

You will see three parsers:

- **StrOutputParser** -> plain Python string
- **JsonOutputParser** -> Python dict
- **PydanticOutputParser** -> validated, typed objects

Why this matters: LLMs generate text. Output parsers turn that text into **reliable data** your code can safely use.

Mental model (what is flowing through the chain)

LangChain v1 treats prompts, models, and parsers as **Runnables**. Each step transforms one object into another:

```
input variables
   -> PromptTemplate / ChatPromptTemplate
   -> PromptValue (messages)
   -> ChatModel
   -> AIMessage
   -> OutputParser
   -> Python object (str / dict / BaseModel)
```

Each parser answers one question: **"Given the model output, how do I turn this into something my code can safely use?"**

Setup: model and environment

We load the API key and initialize a small chat model for consistent examples.

```python
# Load environment variables
import os
from dotenv import load_dotenv

# Pull values from .env into the process env
load_dotenv()

# Import the chat model initializer
from langchain.chat_models import init_chat_model

# Guardrail: ensure the API key is present
if not os.getenv("OPENAI_API_KEY"):
    raise ValueError("OPENAI_API_KEY not found!")
```

```python
# Initialize a lightweight chat model
model = init_chat_model(
    # Small, fast model for demos
```

```
    model="gpt-4o-mini"
)
```

Example 1: StrOutputParser (plain strings)

When to use: When you want the model response as a **plain Python string**, not an AIMessage object.

Intuition (under the hood):

- The prompt formats variables into **message objects** (System/Human).
- The model returns an **AIMessage** (it is not a string yet).
- `StrOutputParser` extracts **only** the `.content` and returns a `str`.

Think of it as the bridge from **LLM world** to **normal Python**.

```
# Import the chat prompt template + string parser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Build a simple prompt with roles
prompt = ChatPromptTemplate.from_messages(
    [
        # System message sets the behavior
        ("system", "You are a concise tutor. Reply in one sentence."),
        # Human message is the actual question
        ("human", "Explain {topic}."),
    ]
)

# Create the simplest parser (AIMessage -> str)
parser = StrOutputParser()
```

```
# Compose the pipeline: prompt -> model -> parser
chain = prompt | model | parser
```

```
# Run the chain with a topic variable
response = chain.invoke(
    {
        "topic": "data leakage"
    }
)

# Confirm the type and value
print(type(response))
print(response)
```

```
<class 'str'>
Data leakage refers to the unintentional exposure of sensitive data to unauthorized individua
ls, often occurring during the data handling or processing stages in machine learning or anal
ytics.
```

Output note: The result is a **Python string**, not an AIMessage.

Example 2: JsonOutputParser (JSON -> dict)

When to use: When you want a dictionary back (so you can do `result["answer"]`).

Intuition (under the hood):

- The model still returns **text**.
- The parser attempts `json.loads(...)` on that text.
- If JSON is valid, you get a **dict**. If not, it fails loudly.

This is the key shift from **best-effort text** to **explicit success or failure**.

```
# Import the JSON parser
from langchain_core.output_parsers.json import JsonOutputParser

# Build a prompt that demands JSON only
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "Return ONLY valid JSON. No markdown, no extra text."),
        (
            "human",
            "Answer the question and also provide a confidence interval score.\n"
            "Question: {question}\n\n"
            "Return JSON with keys: answer (string), confidence (number 0-1)."
        ),
    ]
)

# JSON parser attempts json.loads on the model output
parser = JsonOutputParser()
```

```
# Compose the pipeline and run it
chain = prompt | model | parser

response = chain.invoke(
    {
        "question": "What is overfitting?"
    }
)

# Inspect the parsed dict
print(type(response))
print(response)
print(response.keys())
print("\nanswer =", response["answer"])
print("\nconfidence =", response["confidence"])
```

```
<class 'dict'>
{'answer': "Overfitting is a modeling error that occurs when a machine learning model learns the noise and fluctuations in the training data to the extent that it negatively impacts the model's performance on new data. This means that while the model may perform exceptionally well on the training dataset, it fails to generalize to unseen data, leading to poor predictive performance.", 'confidence': 0.95}
dict_keys(['answer', 'confidence'])

answer = Overfitting is a modeling error that occurs when a machine learning model learns the noise and fluctuations in the training data to the extent that it negatively impacts the model's performance on new data. This means that while the model may perform exceptionally well on the training dataset, it fails to generalize to unseen data, leading to poor predictive performance.

confidence = 0.95
```

Output note: You should see a `dict` with keys like `answer` and `confidence`.

Example 3: PydanticOutputParser (typed objects)

When to use: When you want **validation** and **typed outputs**.

Intuition (under the hood):

- You define a **schema** with Pydantic.
- The parser generates **format instructions** for the model.
- The model returns JSON text.
- LangChain parses JSON and then **validates** it with Pydantic.
- If validation fails, the chain errors early. If it passes, you get a real object.

This turns LLM output into **validated domain objects**, which is much safer for real apps.

```python
# Import Pydantic and the output parser
from pydantic import BaseModel, Field
from langchain_core.output_parsers import PydanticOutputParser

# Build a prompt that will include parser instructions
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a careful tutor. Follow the formatting instructions exactly."),
        (
            "human",
            "Define the term.\n"
            "Term: {term}\n\n"
            "Formatting instructions:\n{format_instructions}"
        ),
    ]
)

# Define the schema we want back
class Definition(BaseModel):
    term: str = Field(description="The term being defined.")
    definition: str = Field(description="A clear definition in plain English.")
    example: str = Field(description="A short example.")

# Create the parser bound to the schema
parser = PydanticOutputParser(pydantic_object=Definition)
```

```python
# Compose and run the chain
chain = prompt | model | parser

response = chain.invoke(
    {
        "term": "regularisation",
        # Use the parser-provided format instructions
        "format_instructions": parser.get_format_instructions()
    }
)

# Inspect the typed result
print(type(response))
print(response)
print("\nterm =", response.term)
print("\ndefinition =", response.definition)
print("\nexample =", response.example)
```

```
<class '__main__.Definition'>
term='regularisation' definition='Regularisation is a technique used in machine learning and statistics to prevent overfitting by adding a penalty term to the loss function, which helps to simplify the model.' example='In linear regression, L2 regularisation (also known as Ridge regression) adds a penalty equal to the square of the magnitude of the coefficients to the loss function.'

term = regularisation

definition = Regularisation is a technique used in machine learning and statistics to prevent overfitting by adding a penalty term to the loss function, which helps to simplify the model.

example = In linear regression, L2 regularisation (also known as Ridge regression) adds a penalty equal to the square of the magnitude of the coefficients to the loss function.
```

Output note: The result is a `Definition` instance with `.term`, `.definition`, `.example`.

Key takeaways

- **StrOutputParser**: best for plain strings.
- **JsonOutputParser**: best for simple structured data as dicts.
- **PydanticOutputParser**: best for validated, typed outputs.

One clean teaching line:

> Output parsers turn an LLM from a text generator into a dependable software component.

If you want the next lesson to flow from here, a great follow-up is: **"What happens when parsing fails, and how do we recover?"**