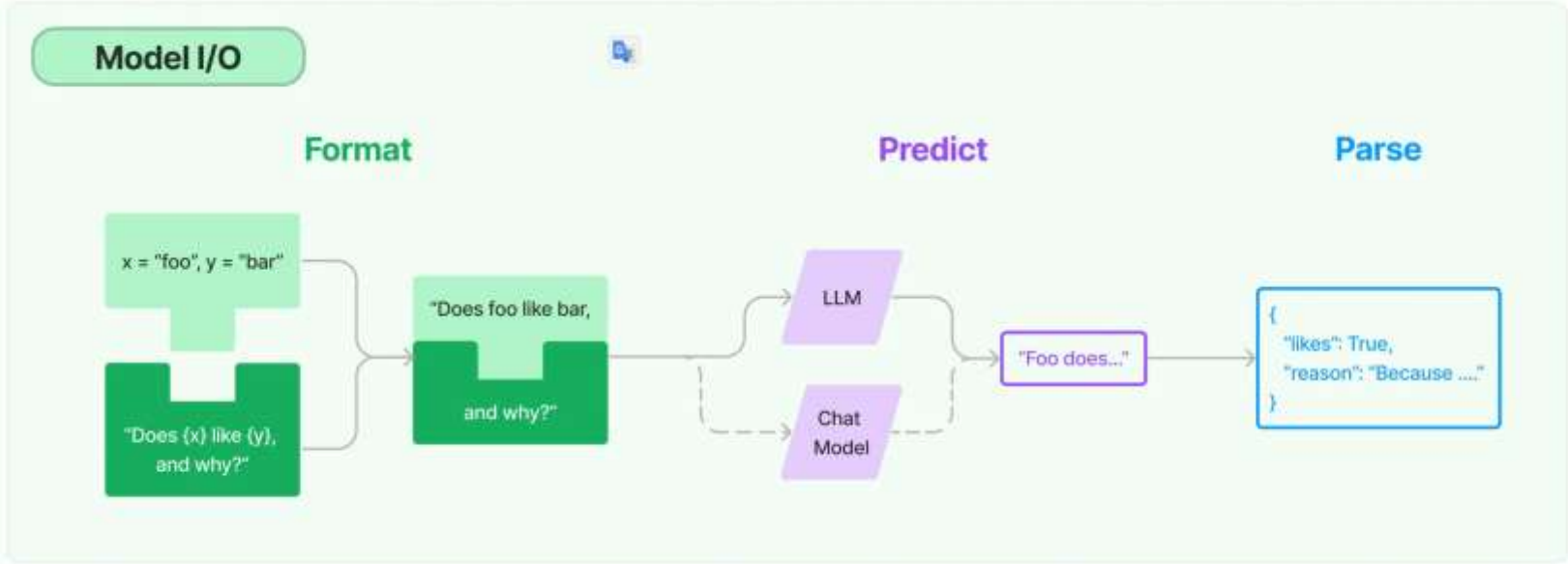
第七章　OutputParserについて

前の章では、いくつかの説明を生成し、それらの説明と理由をCSVファイルに保存しました。この目的を達成するために、プログラムはLLMモデルを呼び出し、構造化出力パーサーやいくつかのデータ処理と保存ツールを利用しました。

この章では、LangChainの出力パーサーについて深く掘り下げ、Pydanticパーサーという新しいパーサーを使って第5章のプログラムを再構築します。この授業は、モデルI/Oフレームワークの補足部分でもあります。



第七章　OutputParserについて

➢　LangChainの出力パーサー

まず、LangChainの出力パーサーが何であるか、そしてどのような種類があるのかを見てみましょう。

言語モデルの出力はテキストであり、これは人間が読むためのものです。しかし、多くの場合、プログラムが処理できる構造化情報が必要になることがあります。これが出力パーサーの役割です。

出力パーサーは、言語モデルの応答を処理し、構築するための専用クラスです。基本的な出力パーサークラスは、通常、2つのコアメソッドを実装する必要があります。

get_format_instructions：このメソッドは、言語モデルの出力をフォーマットするための指示を返す文字列を返す必要があります。これは、出力がどのように組織され、構築されるべきかを示します。
parse：このメソッドは、文字列（言語モデルの出力）を受け取り、それを特定のデータ構造やフォーマットに解析します。このステップは、モデルの出力が私たちの期待に応えていることを確認し、必要な形式で後続処理を行うために使用されます。

第七章　OutputParserについて

- **LangChainの出力パーサー**

さらに、オプションのメソッドもあります。

parse_with_prompt：このメソッドは、文字列（言語モデルの出力）とプロンプト（この出力を生成するためのプロンプト）を受け取り、それを特定のデータ構造に解析します。これにより、元のプロンプトに基づいてモデルの出力を修正または再解析し、出力情報がより正確で要件に合致するようにすることができます。以下は、上記の説明に基づく簡単な擬似コードの例です。

```
class OutputParser:
    def __init__(self):
        pass

    def get_format_instructions(self):
        pass

    def parse(self, model_output):
        pass

    def parse_with_prompt(self, model_output, prompt):
        pass
```

第七章　OutputParserについて

➢　LangChainの出力パーサー

LangChainでは、get_format_instructions、parse、および parse_with_prompt のメソッドを実装することで、さまざまな使用シーンや目的に応じた出力パーサーが設計されています。それでは、一つ一つ見ていきましょう。

リストパーサー（List Parser）：このパーサーは、モデルが生成する出力がリストである必要がある場合に使用されます。例えば、「すべての花の在庫を列挙してください」と尋ねると、モデルの回答はリストになるべきです。

日付時間パーサー（Datetime Parser）：このパーサーは、日付や時間に関連する出力を処理し、モデルの出力が正しい日付または時間のフォーマットであることを保証します。

列挙パーサー（Enum Parser）：このパーサーは、事前に定義された値のセットを処理します。モデルの出力がこれらの事前定義された値のいずれかである必要がある場合に使用します。

第七章　OutputParserについて

➢　LangChainの出力パーサー

構造化出力パーサー（Structured Output Parser）：このパーサーは、複雑で構造化された出力を処理します。アプリケーションがモデルに特定の構造を持つ複雑な回答（例えばレポートや記事など）を生成させる必要がある場合、構造化出力パーサーを使用して実現できます。

Pydantic（JSON）パーサー：このパーサーは、モデルの出力が特定のフォーマットに合ったJSONオブジェクトである必要がある場合に使用します。Pydanticライブラリを使用しており、これはデータ検証ライブラリで、複雑なデータモデルを構築し、モデルの出力が期待されるデータモデルに合致することを保証します。

自動修正パーサー（Auto-Fixing Parser）：このパーサーは、モデルの出力における一般的なエラーを自動的に修正できます。例えば、モデルの出力がテキストであるべきなのに文法やスペルの誤りを含むテキストが返された場合、自動修正パーサーはこれらのエラーを自動的に訂正します。

第七章　OutputParserについて

➢　LangChainの出力パーサー

再試行パーサー（RetryWithErrorOutputParser）：このパーサーは、モデルの初回出力が期待通りでない場合に修正や新しい出力の生成を試みます。例えば、モデルの出力が日付であるべきなのに文字列が返された場合、再試行パーサーはモデルに正しい日付フォーマットを再生成させることができます。

上記の様々なパーサーの中で、前の4つは理解しやすく、使用も直感的です。後の2つは明らかに初心者が取り扱うべきではありません。したがって、次に本章はPydantic（JSON）パーサーについて重点的に説明します。

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

Pydantic（JSON）パーサーは、最も一般的で重要なパーサーの一つです。現在、これを使用して文書生成プログラムを再構築します。

Pydantic は、Python のデータ検証と設定管理ライブラリで、主に Python の型ヒントに基づいています。JSON のために特別に設計されたわけではありませんが、JSON は現代のウェブアプリケーションや API のインタラクションで一般的なデータフォーマットであるため、Pydantic は JSON データの処理と検証に非常に便利です。

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

まず、環境変数を通じてOpenAI APIキーを設定し、その後LangChainライブラリを使用してOpenAIのモデルインスタンスを作成しました。ここでも、言語モデルとしてtext-davinci-003を選択しました。

```
import os
os.environ["OPENAI_API_KEY"] = '你的OpenAI API Key'

from langchain import OpenAI

model = OpenAI(model_name='gpt-3.5-turbo-instruct')
```

第七章　OutputParserについて

- **Pydantic（JSON）パーサーを使用する**

次に、モデルから生成された説明を保存するための空のDataFrameを作成しました。その後、GoodsDescription という名前のPydantic のBaseModel クラスを使用して、期待されるデータフォーマット（つまりデータの構造）を定義しました。

```
import pandas as pd
df = pd.DataFrame(columns=[”goods_type", "price", "description", "reason"])

goods = [”PS5", ”Switch", ”Xbox"]
prices = [”66000", ”32000", ”55000"]

from pydantic import BaseModel, Field
class GoodsDescription(BaseModel):
    goods_type: str = Field(description=“商品の種類")
    price: int = Field(description=“価格")
    description: str = Field(description=“説明文書")
    reason: str = Field(description=“なぜこんな文書")
```

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

ここでは、データフォーマットの検証を担当するPydanticライブラリを使用して、型注釈付きのクラスGoodsDescription を作成しました。このクラスは、自動的に入力データを検証し、入力データが指定した型やその他の検証条件に一致することを保証します。Pydanticには以下のような特徴があります。

データ検証：Pydanticクラスに値を設定する際、データの検証を自動的に行います。例えば、整数が必要なフィールドを作成したが、文字列を設定しようとすると、Pydanticは例外を発生させます。

データ変換：Pydanticはデータ検証だけでなく、データ変換も行います。例えば、整数が必要なフィールドがあり、整数に変換できる文字列（例えば「42」）を提供した場合、Pydanticは自動的にこの文字列を整数42に変換します。

使いやすさ：Pydanticクラスの作成は、普通のPythonクラスを定義するのと同じくらい簡単です。Pythonの型注釈機能を使用することで、クラス定義内で各フィールドの型を指定することができます。

JSONサポート：PydanticクラスはJSONデータから簡単に作成でき、クラスのデータをJSON形式に変換することもできます。

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

次に、出力パーサーを作成し、出力フォーマットの指示を取得します。まず、LangChainライブラリのPydanticOutputParser を使用して出力パーサーを作成しました。このパーサーは、モデルの出力がGoodsDescription のフォーマットに一致するかどうかを検証するために使用されます。その後、パーサーのget_format_instructions メソッドを使用して、出力フォーマットの指示を取得しました。

```
from langchain.output_parsers import PydanticOutputParser
output_parser = PydanticOutputParser(pydantic_object=GoodsDescription)

format_instructions = output_parser.get_format_instructions()

print(format_instructions)
```

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

次に、プログラムは output_parser.get_format_instructions() メソッドを使用して出力フォーマットを生成します。これはPydantic（JSON）パーサーの核心的な価値です。また、これは非常に明確なヒントテンプレートとも言え、モデルに良いガイダンスを提供し、モデルの出力が従うべきフォーマットを説明します。（その中の説明の漢字や仮名はUTF-8エンコードに変換されています。）

この指示は、モデルにJSONスキーマの形式で出力を指示し、有効な出力がどのフィールドを含むべきか、およびそれらのフィールドのデータ型を定義します。例えば、「goods_type」フィールドは文字列型、「price」フィールドは整数型であるべきと指定しています。この指示には、フォーマットが適切な出力の例も提供されています。

次に、この内容をモデルのプロンプトにも伝え、モデルのプロンプトと出力パーサーの要件が一致するようにします。これにより、前後の整合性が取れるようになります。

第七章　OutputParserについて

➢ Pydantic（JSON）パーサーを使用する

次に、モデルに入力プロンプトを生成するためのテンプレートを定義しました。このテンプレートには、モデルが埋める必要のある変数（例えば、価格や商品の種類）と、以前に取得した出力フォーマット指示が含まれています。

```
from langchain import PromptTemplate
prompt_template = """
あなたはプロのビジネスコピーライターです。
価格が {price} 円の {goods_name} について、魅力的な説明を提供していただけますか？
{format_instructions}
 """

prompt = PromptTemplate.from_template(prompt_template,
       partial_variables={"format_instructions": format_instructions})

print(prompt)
```

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

以下は、プロンプトテンプレートの主要なフィールドについて説明します。

input_variables=['goods', 'price']：これはテンプレート内で使用したい入力変数のリストです。ここでは、goodsとpriceの2つの変数を使用しています。後で、具体的な値（例えばPS5、66000円）でこれらの変数を置き換えます。

output_parser=None：これはテンプレート内で使用する可能性のある出力パーサーです。この例では、テンプレート内で出力パーサーを選択せず、モデルの外部で出力を解析するため、ここはNoneです。

partial_variables：テンプレート内で使用したいが、テンプレート生成時にすぐに提供できない変数が含まれています。ここでは、format_instructions を通じて出力フォーマットの詳細な説明を渡しています。

第七章　OutputParserについて

- **Pydantic（JSON）パーサーを使用する**

template：テンプレート文字列そのものです。モデルが生成するテキストの構造が含まれています。この例では、テンプレート文字列は商品説明に関する質問と出力フォーマットに関する説明です。

template_format='f-string'：テンプレート文字列のフォーマットを指定するオプションです。ここではf-stringフォーマットを使用しています。

validate_template=True：テンプレート作成時にテンプレートの有効性をチェックするかどうかを示します。ここでは、テンプレートが有効であることを確認するために、テンプレート作成時にチェックすることを選択しています。

第七章　OutputParserについて

- **Pydantic（JSON）パーサーを使用する**

全体として、このプロンプトテンプレートはモデル入力を生成するためのツールです。テンプレート内で必要な入力変数とテンプレート文字列のフォーマットと構造を定義し、そのテンプレートを使用して各商品に対する説明を生成できます。

その後、実際の情報をプロンプトテンプレートに繰り返し入力し、具体的なプロンプトを生成します。それでは、続けていきましょう。

第七章　OutputParserについて

- **Pydantic（JSON）パーサーを使用する**

この部分はプログラムの本体です。すべての商品とその価格を処理するためにループを使用します。各商品について、プロンプトテンプレートに基づいて入力を作成し、その後モデルの出力を取得します。次に、以前に作成したパーサーを使用してこの出力を解析し、解析された出力をDataFrameに追加します。最後に、すべての結果を表示し、CSVファイルに保存するオプションも選択できます。

```
for good, price in zip(goods, prices):
    input = prompt.format(goods=good, price=price)
    print(input)

    output = model(input)

    parsed_output = output_parser.parse(output)
    parsed_output_dict = parsed_output.dict()

    df.loc[len(df)] = parsed_output.dict()

print(df.to_dict(orient='records'))
```

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

このステップでは、モデルと入力プロンプト（商品種別と価格で構成される）を使用して、具体的な商品の文書要求（フォーマットの説明付き）を生成し、それを大モデルに渡します。つまり、プロンプトテンプレート内の goods と price は具体的な商品に置き換えられ、テンプレート内の {format_instructions} もJSONスキーマで指定されたフォーマット情報に置き換えられます。

次に、プログラムはモデルの出力を解析します。このステップでは、以前に定義した出力パーサー（output_parser）を使用して、モデルの出力を GoodsDescription のインスタンスに解析します。GoodsDescription は以前に定義したPydanticクラスで、商品の種類、価格、説明、および説明の理由が含まれています。

その後、解析された出力をDataFrameに追加します。このステップでは、解析された出力（つまり GoodsDescription インスタンス）を辞書に変換し、その辞書をDataFrameに追加します。このDataFrameはすべての商品説明を保存するために使用します。

第七章　OutputParserについて

➢　Pydantic（JSON）パーサーを使用する

次に、解析された出力をDataFrameに追加します。このステップでは、解析された出力（つまりGoodsDescription インスタンス）を辞書に変換し、その辞書をDataFrameに追加します。このDataFrameはすべての商品の説明を保存するために使用します。

要するに、Pydanticの利点は解析が容易であり、解析後の辞書形式のリストはデータ分析、処理、および保存が非常に便利です。各辞書は1つのレコードを表し、そのキー（例えば goods、price、description、reason）はフィールド名で、対応する値はそのフィールドの内容です。これにより、各フィールドは1列に対応し、各辞書は1行になります。これらはDataFrame形式で表現し、処理するのに適しています。

第七章　OutputParserについて

➢　まとめ

構造化パーサーとPydanticパーサーは、どちらも大規模言語モデルからフォーマットされた出力を取得することを目的としています。構造化パーサーはシンプルなテキストレスポンスに適しており、Pydanticパーサーは複雑なデータ構造と型のサポートを提供します。どちらのパーサーを選択するかは、アプリケーションの具体的なニーズと出力の複雑さに依存します。

未取り扱いの内容について、自動修正パーサーは主に小さなフォーマットエラーを修正するのに適しており、原則として「受動的」で、元の出力に問題が発生した場合にのみ修正を行います。一方、リトライパーサーはフォーマットエラーや内容の欠如など、より複雑な問題を処理できます。モデルと再度やり取りすることで、出力をより完全で期待通りのものにします。

どのパーサーを選択するかは、具体的なアプリケーションシナリオを考慮する必要があります。