2章 Model I/O 言語モデルを扱いやすくする

01 言語モデルを使ったアプリケーションの基本的なしくみを理解する

#チャットボット／#言語モデル

LangChainの最も基本的なモジュールであるModel I/Oは言語モデルを呼び出す方法を提供します。 具体的にどのようなことができるのかコードを書きつつ見ていきましょう

言語モデルを呼び出すとは

改めて「言語モデルを呼び出す」とはどういうことでしょうか？

ChatGPTなどWebサービスでは、テキストボックスにメッセージを入力し、送信ボタンをクリックすることで結果が出力されます。

これはテキストボックスに入力したメッセージから言語モデルを呼び出しているといえるでしょう。

このように言語モデルを呼び出す際に、入力となるテキストのことを「プロンプト」と呼びます。今後は言語モデルの入力となるテキストはプロンプトと呼ぶので覚えておきましょう。

言語モデルを使ったアプリケーションを作るときにはこの呼び出しをPythonなどで作成したプログラムから行います。前の章で確認しましたが、例としてLangChainを使わないでOpenAIの言語モデルである「gpt-3.5-turbo」を呼び出すコードはどのようなものか見ていきましょう。

以下コードは実際に実行する必要はありません。

・sample.py

```
001 import openai      OpenAIが用意しているPythonパッケージをインポートする
002
003 response = openai.ChatCompletion.create(      OpenAIのAPIを呼び出すことですことで、言語モデルを呼び出している
004     model="gpt-3.5-turbo",      呼び出す言語モデルの名前
005     messages=[
006         {
```

```
007             "role": "user",
008             "content": "iPhone8のリリース日を教えて"      入力する文章(プロンプト)
009         },
010     ]
011 )
012 print(response)      結果を表示
013
```

上記のソースコードでは4行目で設定されている「gpt-3.5-turbo」という言語モデルを「iPhone8のリリース日を教えて」というテキスト、「user」というロール（役割）でopenaiが用意しているパッケージを使って呼び出しています。

単純なアプリケーションなら上記のようなソースコードで問題ありませんが、実際に言語モデルを使ったアプリケーションを開発する際には問題になることがよくあります。 言語モデルを使ったアプリケーションは、すべて手続き型で作成する従来のアプリケーションとは異なり、よい結果を得るためには試行錯誤が必要です。

まず、8行目の「iPhone8のリリース日を教えて」というプロンプトです。言語モデルから得られる結果は入力されるプロンプトの書き方で異なります。

たとえば「iPhone8のリリース日を教えて」と入力した場合に、「2017/09/22」と出力されるか「2017年9月22日」と出力されるかはわかりません。しかし「iPhone8のリリース日をyyyy/mm/ddという形式で教えて」と入力することで出力される結果を固定し、求める結果を出力させやすくすることが可能です。

また、4行目では「gpt-3.5-turbo」とモデル名が指定されていますが、より長いテキストを処理できる「gpt-3.5-turbo-16k」に差し替えたいなら以下のようにモデル名を書き換えるだけで問題ありません。

・sample.py

```
001 ~~~~中略~~~~
002     model="gpt-3.5-turbo",      呼び出す言語モデルの名前
003 ~~~~中略~~~~
004
```

しかし言語モデルはOpenAIのGPT-3.5やGPT-4だけではありません。AnthropicのClaude2を使って結果がどのように変わるかみたい場合もあるでしょう。この場合、先ほどのソースコードはOpenAIにしか対応していないので、ほぼすべてのソースコードを書き直す必要があります。

このように言語モデルを呼び出すプロンプトを試行錯誤して書き換えたり、モデルを差し替えたりするのはとても手間がかかります。Model I/Oモジュールではこのような手間を減らすための手段を提供しています。また、後に紹介するほかのモジュールでもModel I/Oと組み合わせる必要があるものが多いのでしっかりと学んでいきましょう。

Model I/OはLangChainで最も基本的なモジュール

Model I/Oモジュールは単体でも使用できますが、実際のアプリケーションを開発する際にはこのモジュールだけですべて作ることは現実的に難しく、ほかのモジュールと組み合わせて使用することが一般的です。たとえば、Promptsモジュールはプロンプトを最適化するために使用されるだけでなく、後に紹介するChainsモジュールなどでも使われており、Language Modelsはほぼすべてのモジュールで使用することになります。

提供している機能は単純なものが多いですが、非常に重要なモジュールなのでどのように使うのかしっかり学んでいきましょう。

Model I/Oを構成する3つのサブモジュール

LangChainのすべてのモジュールはサブモジュールを持っています。Model I/Oモジュールも例外ではなく、3つのサブモジュールから構成されています。ここではざっとどんな機能か見てみましょう。詳しくは後ほど解説します。

①Prompts

Promptsモジュールは言語モデルを呼び出すためのプロンプトを構築するのに便利な機能を提供します。用途によってさまざまな孫モジュールが用意されています。たとえばプロンプトと変数を組み合わせたり、大量の例示を効率的にプロンプトに挿入したりできます。 さまざまな処理をして、求めるプロンプトを作成しやすくするのが目的です。

②Language Models

Language Modelsモジュールは、さまざまな言語モデルを同一のインターフェースで呼び出すための機能を提供します。OpenAIのモデルだけでなく、Anthropicの

Claudeなどほかのモデルも同じように呼び出せます。これにより、異なるモデルを試す際に、既存のコードを一から書き直す必要がなくなります。

③Output parsers

Output parsersモジュールは、言語モデルから得られる出力を解析し、アプリケーションで利用しやすい形に変換するための機能を提供します。出力文字列を整形したり、特定の情報を抽出したりするために使用します。このモジュールにより、出力を構造化したデータとして扱うことが容易になります。

ここからは実際にコードを書きつつ各モジュールの動きを見ていきましょう。

Language Modelsを使ってgpt-3.5-turboを呼び出す

実際にLanguage Modelsモジュールを使ってOpenAIのChatモデルであるgpt-3.5-turboを呼び出してみましょう。

まずは1ページの「1-6-1 Pythonの実行環境を整える」で作成したディレクトリへ移動し「02_model_io」という名前で新規ディレクトリを作成しましょう。 作成したディレクトリをVSCodeで開いてください。

［ファイル］メニューの［新規ファイル］から、「language_modules.py」というファイルを作成し、以下の通りに入力してください。

・language_modules.py

```
001 from langchain.chat_models import ChatOpenAI      モジュールをインポート
002 from langchain.schema import HumanMessage      ユーザーからのメッセージである
HumanMessageをインポート
003
004 chat = ChatOpenAI(      クライアントを作成しchatへ保存
005     model="gpt-3.5-turbo",      呼び出すモデルを指定
006 )
007
008 result = chat(      実行する
009     [
010         HumanMessage(content="こんにちは！"),
011     ]
012 )
013 print(result.content)
014
```

次にPythonで上記コードを実行します。

```
python3 language_modules.py
```

すると、以下のような結果が確認できます。なお、生成結果はまったく同じになるとは限りません。

```
こんにちは！私はAIアシスタントです。何かお手伝いできますか？
```

コードの要点を見ていきましょう。

・language_modules.py

```
from langchain.chat_models import ChatOpenAI    モジュールをインポート

~~~省略~~~

chat = ChatOpenAI(  クライアントを作成しchatへ保存
    model="gpt-3.5-turbo",    呼び出すモデルを指定
)
```

まず1行目でLanguage Modelsの1つであるChatOpenAIクラスをインポートしています。ChatOpenAIクラスはOpenAIのChatモデルを呼び出す際に使用されます。実際に5行目ではOpenAIのChatモデルの1つであるgpt-3.5-turboを指定しています。

・language_modules.py

```
from langchain.schema import HumanMessage    ユーザーからのメッセージである
HumanMessageをインポート

~~~省略~~~

result = chat(  実行する
    [
        HumanMessage(content="こんにちは！"),
    ]
)
```

```
print(
    result.content
)
```

10行目ではHumanMessageをcontentに言語モデルへ送信したい内容を入力し初期化しています。HumanMessageは人間からのメッセージあることを表しており、contentはその内容を表します。これらのHumanMessageを使って11行目で言語モデルを呼び出すことで、入力されたメッセージをもとに言語モデルを呼び出せます。

AIMessageを使って言語モデルからの返答を表すことができる

LangChainでは対話形式のやりとりを表現するために、AIMessageも用意されています。たとえば、最初に「茶碗蒸しの作り方を教えて」と問い合わると言語モデルからレシピが返されるはずです。 このレシピを英語に翻訳したいときには「英語に翻訳して」と指示することで英語に翻訳されたレシピを受け取ることができます。このような会話の流れをAIMessageを使ってどのように対話形式のやりとりを表現するのか見てみましょう。

以下コードは説明を意図したもので実際に実行する必要はありません。

・language_modules_ai_message_sample.py

```
result = chat(  実行する
    [
        HumanMessage(content="茶碗蒸しの作り方を教えて"),
        AIMessage(content="{ChatModelからの返答である茶碗蒸しの作り方}"),
        HumanMessage(content="英語に翻訳して"),
    ]
)

```

このようにChatModelではHumanMessage、AIMessageを使用することで言語モデルとの対話形式のやりとりを表現できます。

Language Modelsのみでこのように過去の返答を踏まえた解答させるには都度ソースコードの書き換えが必要になり、非常に面倒で、対話を用いたアプリケーション開発は難しいでしょう。LangChainではこのような対話をサポートするためのMemoryというモジュールが用意されています（2ページで解説）。

SystemMessageを使って言語モデルの人格や設定を定義する

また、こういった対話機能をカスタマイズできるSystemMessageも用意されています。これは対話を表現するものではなく、言語への直接的な指示を書くことができます。たとえば言語モデルの人格や設定などを入力することで、返答の文体をよりフランクなものに変更できます。

SystemMessageを設定して返答の文体などを変更する方法を見てみましょう。以下コードは説明を意図したもので実際に実行する必要はありません。

・language_modules_system_message_sample.py

```
result = chat(
    [
        SystemMessage(content="あなたは親しい友人です。返答は敬語を使わず、フランクに会話してください。"),  システムメッセージを使用して設定を追加
        HumanMessage(content="こんにちは！"),
    ]
)

```

これを実行すると以下のような結果が返ってきます。

```
やぁ、こんにちは！元気してる？
```

SystemMessageに入力した指示の通りに、文体をフランクなものに変更できました。

point　ChatModelは差し替えることができる

Language Modelsは共通のインターフェイスを持ち、簡単に差し替えることができると説明しました。今回はOpenAIの対話形式の言語モデルを読み込むための「ChatOpenAI」を使用しましたが、これをOpenAIではなく、Anthropicが開発した言語モデルに差し替える場合にはどのように変更するのか見てみましょう。

Anthropicの言語モデルをAPI経由で使用するには、執筆時点では申請と審査が必要になりますが、ここでは一例として紹介します。

Anthropicが開発する対話形式の言語モデルは「ChatAnthropic」で使用できま

す。つまり先ほどのコードを以下のように編集するだけで言語モデルだけを差し替えることが可能になります。

・language_modules_chat_anthropic_sample.py

```
from langchain.chat_models import ChatAnthropic    AnthropicのChatModelをインポートするように変更

~~~省略~~~

chat = ChatAnthropic()    ChatAnthropicのLanguage Modelsを初期化
~~~省略~~~
```

このように同じ「ChatModel」であれば簡単に対話形式の言語モデルの差し替えを行うことが可能になります。

変数をプロンプトに展開する

言語モデルをプログラムから呼び出す場合、用意してあるプロンプトとPythonからの入力を組み合わせることがよくあります。

Promptsモジュールの最も基本的なモジュールであるPromptTemplateを使ってPythonからの入力とプロンプトを組み合わせてみましょう。

VSCodeの［ファイル］メニューの［新しいテキストファイル］から「prompt.py」というファイルを作成し、以下の通りに入力してください。

・prompt.py

```
001 from langchain import PromptTemplate    PromptTemplateをインポート
002
003 prompt = PromptTemplate(    PromptTemplateを初期化する
004     template="{product}はどこの会社が開発した製品ですか？",    {product}という変数を含むプロンプトを作成する
005     input_variables=[
006         "product"    productに入力する変数を指定する
007     ]
008 )
009
010 print(prompt.format(product="iPhone"))
```

```
011 print(prompt.format(product="Xperia"))
```

次にVSCodeの［ターミナル］メニューから［新しいターミナル］を選択してターミナルを開き、Pythonで上記コードを実行します。

```
python3 prompt.py
```

すると、以下のような結果が確認できます。

```
iPhoneはどこの会社が開発した製品ですか？
Xperiaはどこの会社が開発した製品ですか？
```

PromptTemplateを使ってプロンプトを生成できることが確認できました。
このPromptTemplateを使用するためには以下２つのステップが必要です。

1. PromptTemplateの準備
2. 準備したPromptTemplateを使用する

まずは、PromptTemplateの準備です。３行目でPromptTemplateを初期化してtemplateとinput_variablesを引数に入れ、結果をprompt変数に保存しました。

・prompt.py

```
~~省略~~
prompt = PromptTemplate(      PromptTemplateを初期化する
    template="{product}はどこの会社が開発した製品ですか？",      {product}という変数を含むプロンプトを作成する
    input_variables=[
        "product"      productに入力する変数を指定する
    ]
)
~~省略~~
```

templateにはもととなるテンプレートをテキストで入力します。「{product}はどこの会社が開発した製品ですか？」のように{}で置き換えたい名前を囲みます。 そしてinput_variablesには置き換えたい名前を配列で入力します。テンプレートには{product}という文字列があり、これを後で置き換えたいことを意図しているので、

ここでは"product"を配列として入力しています。

以上で準備は完了です。次に準備したPromptTemplateを使用する方法を見ていきましょう。

・prompt.py

```
001 ~~省略~~
002 print(prompt.format(product="iPhone"))
003 print(prompt.format(product="Xperia"))
```

10行目ではprompt.format(product="iPhone")を実行しています。ここではpromptを使用し、formatメソッド実際のプロンプトを生成、つまり、テキストを生成しています。結果、以下のようなプロンプトを生成できました。

```
iPhoneはどこの会社が開発した製品ですか？
```

11行目ではこのPromptTemplateを使うための名前として"product"を入力しています。そしてこの結果は、以下のように先ほどとは違うプロンプトが生成できました。

```
Xperiaはどこの会社が開発した製品ですか？
```

PromptTemplateでプロンプトを生成できることを確認できました。

PromptTemplateに用意されているその他の機能

PromptTemplateにはほかにも便利な機能が用意されています。 今回紹介したコードでは以下のような形でproductを渡していました。

```
prompt.format(product="Xperia")
```

では、formatを呼び出すときに以下のようにproductを入力しない場合はどのような動きになるでしょうか。

・prompt.py

```
~~省略~~
print(prompt.format())
```

　すると以下のようなエラーが表示されます。

```
KeyError: 'product'
```

　これはproductがinput_variablesで必要な入力として定義されているにもかかわらず、プロンプトを生成しようとしていることで発生しています。

　プロンプトは結局のところはただのテキストです。ただのテキストである以上、どのような方法でも作成できてしまいます。しかし、実際システムに組み込む際にはプロンプトも強い制約をもって生成することにより、安定したアプリケーションを作成できるようになります。このようにPromptTemplateは、プロンプトを作成するために再現可能な方法を提供します。

Language ModelsとPromptTemplateを組み合わせる

　次は3ページの「Language Modelsを使ってgpt-3.5-turboを呼び出す」で作成した「Language Models」を呼び出すコードとPromptTemplateを組み合わせてみましょう。［ファイル］メニューの［新しいテキストファイル］から、「prompt_and_language_model.py」というファイルを作成し、以下の通りに入力してください。

・prompt_and_language_model.py

```
001 from langchain import PromptTemplate
002 from langchain.chat_models import ChatOpenAI
003 from langchain.schema import HumanMessage
004
005 chat = ChatOpenAI(      クライアントを作成しchatへ保存
006     model="gpt-3.5-turbo",      呼び出すモデルを指定
007 )
008
009 prompt = PromptTemplate(      PromptTemplateを作成する
010     template="{product}はどこの会社が開発した製品ですか？",      {product}
```

```
という変数を含むプロンプトを作成する
    input_variables=[
        "product"      productに入力する変数を指定する
    ]
)

result = chat(   実行する
    [
        HumanMessage(content=prompt.format(product="iPhone")),
    ]
)
print(result.content)

```

今回のコードでは、PromptTemplateで生成したプロンプトをLanguage Modelsを使って呼び出しています。

次にPythonで上記コードを実行します。

```
python3 prompt_and_language_model.py
```

すると以下のような結果を確認できます。

```
iPhoneはアメリカのApple Inc.（アップル）が開発した製品です。
```

それでは今回作成したコードを詳しく見ていきましょう。

まずは、5行目でOpenAIの「Chat」モデルである「gpt-3.5-turbo」をChatOpenAIで初期化し、次に9行目で先ほどと同じようにPromptTemplateを初期化しています。

18行目では、prompt.formatをproductにiPhoneを入力して実行することでプロンプトを作成しています。16行目で実行し、21行目で結果を表示しています。

これで、PromptTemplateを使って変数とプロンプトを組み合わせて実行できることを確認できました。このようにLangChainでは複数存在するモジュールを作成しつつアプリケーションを作成していきます。

PromptTemplateの初期化方法の種類

本書ではPromptTemplateを初期化する際には以下のようにクラスを初期化する

方法をとっています。

・prompt_and_language_model.py

```
001 prompt = PromptTemplate(      PromptTemplateを作成する
002     template="{product}はどこの会社が開発した製品ですか？",      {product}
という変数を含むプロンプトを作成する
003     input_variables=[
004         "product"      productに入力する変数を指定する
005     ]
006 )
```

PromptTemplateは上記以外にもいくつか初期化する方法が存在します。

たとえば、以下のようにinput_variablesを直接指定せず、テンプレートから直接初期化することもできます。

・prompt_template_from_template_sample.py

```
001 prompt = PromptTemplate.from_template("{product}はどこの会社が開発した
製品ですか？")
```

単にPromptTemplateを初期化するだけならこのほうが短く書けますが、本書ではわかりやすさのためにinput_variablesも指定する書き方に統一しています。

また、jsonファイルに保存したプロンプトを読み出す方法も存在します。以下のように「prompt_template_from_template_save_sample.py」を作成して、Pythonで実行します。

・prompt_template_from_template_save_sample.py

```
001 from langchain.prompts import PromptTemplate
002
003 prompt = PromptTemplate(template="{product}はどこの会社が開発した製品
ですか？", input_variables=["product"])
004 prompt_json = prompt.save("prompt.json")      PromptTemplateをJSONに変換する
005
```

すると以下のようなJSONが作成されます。

```
{
    "input_variables": [
        "product"
    ],
    "output_parser": null,
    "partial_variables": {},
    "template": "
{product}\u306f\u3069\u3053\u306e\u4f1a\u793e\u304c\u958b\u767a\u3057\u305
f\u88fd\u54c1\u3067\u3059\u304b\uff1f",
    "template_format": "f-string",
    "validate_template": true,
    "_type": "prompt"
}
```

このJSONを以下のようにファイルから読み出すことでPromptTemplateを作成できます。

・prompt_template_from_template_load_sample.py

```
001 from langchain.prompts import load_prompt
002
003 loaded_prompt = load_prompt("prompt.json")  JSONからPromptTemplateを読み込む
004
005 print(loaded_prompt.format(product="iPhone"))  PromptTemplateを使って文章を生成する
006
```

このようにPromptTemplateをjsonファイルとして保存することで、さまざまな活用ができます。

たとえば、saveメソッドでユーザーが操作するアプリケーションのプロンプトをあらかじめjsonファイルに保存しておき、保存されたjsonファイルを管理者のみが操作できる管理画面で更新し、上書きできるようにすれば、ソースコードの編集をすることなくプロンプトの編集が可能になります。

このようにPromptTemplateはさまざまな方法で生成できます。目的にあった方法で生成しましょう。

リスト形式で結果を受け取る

最後にOutput Parsersを使って言語モデルから受け取った結果を構造化してみましょう。 言語モデルを呼び出して得られる結果はテキスト形式になります。しかし、言語モデルの呼び出し結果をプログラムから使いたい場合にはリスト形式などで構造化されたデータを受け取りたい場合があります。 Output Parserはこの言語モデルの呼び出し結果の構造化を行います。

では、prompt_and_language_model.pyをもとに結果をリスト形式で受け取ってみましょう。［ファイル］メニューの［新しいテキストファイル］から、「list_output_parser.py」というファイルを作成し以下を入力してください。

・list_output_parser.py

```
001 from langchain.chat_models import ChatOpenAI
002 from langchain.output_parsers import \
003     CommaSeparatedListOutputParser  Output Parserである
CommaSeparatedListOutputParserをインポート
004 from langchain.schema import HumanMessage
005
006 output_parser = CommaSeparatedListOutputParser()
CommaSeparatedListOutputParserを初期化
007
008 chat = ChatOpenAI(model="gpt-3.5-turbo", )
009
010 result = chat(
011     [
012         HumanMessage(content="Appleが開発した代表的な製品を3つ教えてく
ださい"),
013         HumanMessage(content=output_parser.get_format_instructions()),
  output_parser.get_format_instructions()を実行し、言語モデルへの指示を追加する
014     ]
015 )
016
017 output = output_parser.parse(result.content)  出力結果を解析してリスト形式
に変換する
018
019 for item in output:  リストを一つずつ取り出す
020     print("代表的な製品 => " + item)
021
```

次にPythonで上記コードを実行します。

```
python3 model_io.py
```

すると、以下のような結果が確認できます。

```
代表的な製品 => iPhone
代表的な製品 => Macbook
代表的な製品 => iPad
```

どのような動きになっているのか詳しく見ていきましょう。

CommaSeparatedListOutputParserは結果をリスト形式で受け取るOutputParserです。 6行目で、CommaSeparatedListOutputParserを初期化し、output_parser変数へ保存し、後で使うための準備をしています。

CommaSeparatedListOutputParserで行われる処理は以下の2つになります。

- リスト形式で出力するように、言語モデルへ出力形式の指示の追加
- 出力結果を解析し、リスト形式に変換

まず、リスト形式で出力するように、言語モデルへ出力形式の指示を追加する処理は13行目で行われます。

output_parser.get_format_instructions()を実行すると、以下のようなプロンプトが確認できます。

```
Your response should be a list of comma separated values, eg: `foo, bar,
baz`
```

翻訳すると以下になり、言語モデルへ出力形式の指示を追加していることがわかります。

```
応答は`foo, bar, baz`のようなカンマで区切られた値のリストでなければなりま
せん。
```

つまり、言語モデルは「Appleが開発した代表的な製品を3つ教えてください」と、「応答はfoo, bar, bazのようなカンマで区切られた値のリストでなければなり

ません。」という２つのプロンプトを使って呼び出されることになり、言語モデルへの指示に加えて出力形式の指示も追加されていることがわかります。

10行目では先ほどのプロンプトを使って言語モデルを呼び出し、結果をresultで受け取っています。

次に17行目ではoutput_parser.parse()で出力結果を解析し、言語モデルからの応答をリスト形式に変換しています。

今回の例では「["iPhone", "Macbook", "iPad"]」という**文字列**が言語モデルから返され、これをoutput_parser.parse()で実行することにより、Pythonの**配列**へと変換されています。

19行目ではリスト形式に変換されたことで、python上でforを実行することが確認できます。

このように言語モデルが生成する出力は、デフォルトではプレーンテキストの文字列です。この文字列をそのまま利用することもできますが、アプリケーションを開発する場合、この文字列から特定の情報を抽出したり、データとして構造化することが多く必要となります。

アプリケーションで利用しやすいデータに変換するためには、出力文字列を解析して必要な情報を抽出する処理が不可欠です。解析された構造化データは、データベースに保存したりほかのAPIに渡したりといった後続の処理で簡単に利用できます。一方、プレーンテキストのままでは、文字列処理を駆使してデータを抽出する必要が生じ、コードが複雑になりやすいです。また、出力内容が不完全であった場合にエラー処理をしやすくするためにも、データとして解析して構造化することが重要です。たとえば、必須の項目がない場合にエラーを出力するといったバリデーションが可能になります。

LangChainが提供するOutput Parsersを利用することで、望みのデータ構造に合わせて自由にパース（変換）でき、アプリケーションの要件に即した解析処理を実装できます。

このように、出力を解析することで、単なるテキストから意味のあるデータへと変換し、アプリケーションでの利用を容易にすることができます。Output Parsersは素早くパース処理を実装するために強力なツールといえます。

column　foo, bar, bazってなに？

「foo」、「bar」、「baz」は、プログラミングの世界で一般的に使われる仮の名前です。サンプルプログラムなどで、まったく意味のない変数名や関数名をつけるときなどに使わます。 もし適当にbookやcupなどの意味のあ

る名前をつけてしまうと、読者やほかのプログラマーはその名前が何か特定の目的を果たすかのように誤解する可能性があります。たとえば、'book'という名前の変数があると、それが何かの書籍に関するデータを保持していると解釈されることが一般的です。しかし、サンプルコードや教材では、その変数が実際に何を示しているのかは重要ではない場合が多いです。

そこで、「foo」、「bar」、「baz」のような意味のない名前が使われます。これらの名前はプログラミングにおけるメタ構文変数（プログラム内で具体的な機能や役割を持つことを期待しない変数）として広く認識されています。これにより、読者は変数名自体に注目することなく、プログラムの構造やロジックに集中できます。

02 Language Models -モデルを使いやすく

#言語モデル／#LanguageModel/#ChatModel/#LLM
前のセクションでModel I/Oを使った開発の一連の流れを確認しました。このセクションでは言語モデルを扱うモジュールであるLanguage Modelsについて確認しましょう。

統一されたインターフェイスで使いやすく

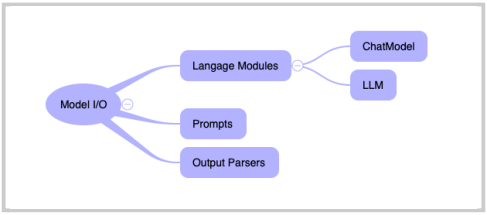
Model I/Oモジュールのサブモジュールである Language Modelsの目的は、さまざまな種類がある言語モデルを統一したインターフェイスを使って扱いやすくすることです。 OpenAI社が開発する言語モデルだけとっても、gpt3.5-turboとgpt-3.5-turbo-instructでは呼び出し方が異なります。アプリケーションを開発する過程で、プロンプトやモデルを差し替えてみたりと試行錯誤することは多くあります。

このように試行錯誤をしているときに、それぞれのモデルで異なる呼び出し方について調べつつ作業するのは手間がかかるということはイメージできると思います。このモジュールを使うことで細かい呼び出し先のURLや使い方を調べることなく統一された方法でアクセスできるようになります。

ChatModelとLLM

Language Modelsには使用する言語モデルに合わせて大きく分けて2種類のモジュール（Model I/Oの孫モジュール）が用意されています。OpenAIの「Chat」モデルのような対話形式で使用する言語モデルを扱う「ChatModel」、OpenAIの「Complete」のような文章の続きを用意する言語モデルを扱う「LLM」です。

要作図



ChatModelとLLMの違いは前提となる入力と出力です。ChatModelは一連の対話

（HumanMessageやAIMessageの配列）を入力とし、次の返答を予測します。対話形式のテキスト生成、特にチャットボットの開発に適しています。ChatModelは前のメッセージのコンテキストを考慮するため、全体の対話の流れを理解しやすい特性を持っています。

一方、LLMは対話ではなく、文の続きを予測します。このモデルは1つのプロンプトだけを考慮します。

今回はLLMを使ってテキストの続きを予測してみましょう。［ファイル］メニューの［新しいテキストファイル］から、「model_io_llm.py」というファイルを作成し、以下の通りに入力してください。

・model_io_llm.py

```
from langchain.llms import OpenAI

llm = OpenAI(model="gpt-3.5-turbo-instruct"    呼び出すモデルを指定
             )

result = llm(
    "美味しいラーメンを",    言語モデルに入力されるテキスト
    stop="。"    「。」が出力された時点で続きを生成しないように
)
print(result)

```

次にPythonで上記コードを実行します。

```
python3 model_io_llm.py
```

すると、以下のような結果が確認できます。

```
食べたいです
```

OpenAIクラスはOpenAIの続きを生成することが目的のLLMなので、「美味しいラーメンが」に続く「食べたいです」といった文章が出力されました。

LLMは1章の4ページで紹介した、OpenAIの「Complete」モデルをLangChainから使う場合に使用します。ChatModelの場合と同様に同じLLMモジュールなら簡単に差し替えできます。

ローカルで実行可能な言語モデルである、GPT4Allに差し替えるには以下のように変更するだけです。

・model_io_llm.py

```
001 from langchain.llms import GPT4All    読み込むLLMをGPT4Allに変更する
002
003 llm = GPT4All()    GPT4AllのLanguage Modelsを初期化
004 ~~~省略~~~
```

LangChainでほかのモジュールとLanguage Modelsを組み合わせて使うときに、LLMを前提とするもの、ChatModelを前提とするものそれぞれが存在します。 モジュールを使うときにどちらを使っているか把握しておく必要があるので違いを覚えておきましょう。

Language Modelsの便利な機能

Language Modelsには「ChatModel」、「LLM」が存在し、差し替えられることについて見てきましたが、Language Modelsでできることはこれだけではありません。具体的に見ていきましょう。

キャッシュをかけることができる

OpenAIなどのAPIは~ページで解説したとおり、使用したトークン数により課金されます。たとえば、同じプロンプトを2回送信すると2回分の料金がかかってしまいます。また、当然2回APIを呼び出すことになり、実行時間が倍かかることになり効率がよくありません。Language Modelsではこのような問題を解決するために簡単にキャッシュをかけることができる機能が用意されています。

実際にどのように動くか見ていきましょう。［ファイル］メニューの［新しいテキストファイル］から、「chat_model_cache.py」というファイルを作成し、以下の通りに入力してください。

・chat_model_cache.py

```
001 import time    実行時間を計測するためにtimeモジュールをインポート
002 import langchain
003 from langchain.cache import InMemoryCache    InMemoryCacheをインポート
```

```
004 from langchain.chat_models import ChatOpenAI
005 from langchain.schema import HumanMessage
006
007 langchain.llm_cache = InMemoryCache()  llm_cacheにInMemoryCacheを設定
008
009 chat = ChatOpenAI()
010 start = time.time()  実行開始時間を記録
011 result = chat([  一度目の実行を行う
012     HumanMessage(content="こんにちは！")
013 ])
014
015 end = time.time()  実行終了時間を記録
016 print(result.content)
017 print(f"実行時間: {end - start}秒")
018
019 start = time.time()  実行開始時間を記録
020 result = chat([  同じ内容で二度目の実行を行うことでキャッシュが利用され、即時に
実行完了している
021     HumanMessage(content="こんにちは！")
022 ])
023
024 end = time.time()  実行終了時間を記録
025 print(result.content)
026 print(f"実行時間: {end - start}秒")
027
```

入力が完了したら以下のコマンドで実行します。

```
python3 chat_model_cache.py
```

すると以下のような表示が確認できます。

```
こんにちは！いつもお世話になっています。どのようなご用件でしょうか？
実行時間: 1.7952373027801514秒
こんにちは！いつもお世話になっています。どのようなご用件でしょうか？
実行時間: 0.0007660388946533203秒
```

7行目では langchain.llm_cache に InMemoryCache() を設定しています。InMemoryCache とは、メモリ内にデータを一時的に保持するキャッシュ方法を提

供するクラスです。特定の要求に対する応答が一度生成されると、それはキャッシュに保存され、同じ要求が再度行われたときには、すでに保存されている応答をすぐに提供することが可能になります。この結果、時間とリソースの節約につながります。

ただしメモリ内のキャッシュは、プログラムが実行されている間は保持されますが、プログラムが終了すると削除されます。今回の場合はプログラムの実行中、つまり以下コマンドの実行開始から終了までは保持されますが、もう一度実行するとキャッシュは削除されてしまいます。

```
python3 chat_model_cache.py
```

長期間にわたってキャッシュする必要がある場合や、プログラムの再起動をまたいでキャッシュを保持したい場合には、InMemoryCacheではなくSQLiteというデータベースに保存できるSQLiteCacheなどを使うとよいでしょう。

今回の実行例では、"こんにちは！"というメッセージに対する応答を初めて生成するのに約1.8秒かかりましたが、同じメッセージに対する応答を再度生成するのには、キャッシュを利用してほぼ瞬時にできました。これにより、APIの呼び出し回数とそれに伴う課金を減らせます。

Language Modelsで簡単にキャッシュをかけ、動作を高速化するための便利な機能の動作を確認できました。

結果を逐次表示させる

Language Modelsの機能の1つに、実行中の処理を逐次表示させるstreaming機能があります。

逐次表示とは、処理が完了する前に一部の結果を順次受け取り、表示することです。この機能は、長い応答を生成する場合や、ユーザーに対してリアルタイムな返答を提供したい場合に役立ちます。

LangChainでは、このstreaming機能を利用するために、callbackという仕組みを提供しています。callbackは、特定の処理が発生したときに実行される関数やクラスを指定できます。これにより、自分のプログラムが必要とする任意の処理を組み込むことが可能になります。

LangChainのstreaming機能とcallbackを使用して、APIの実行中に逐次結果を表示する機能を作成してみましょう。［ファイル］メニューの［新しいテキストファイル］から、「chat_model_streaming.py」というファイルを作成し、以下の通りに入力してください。

・chat_model_streaming.py

```
001 from langchain.callbacks.streaming_stdout import
StreamingStdOutCallbackHandler
002 from langchain.chat_models import ChatOpenAI
003 from langchain.schema import HumanMessage
004
005 chat = ChatOpenAI(
006     streaming=True,  streamingをTrueに設定し、ストリーミングモードで実行
007     callbacks=[
008         StreamingStdOutCallbackHandler()  StreamingStdOutCallbackHandler
をコールバックとして設定
009     ]
010 )
011 resp = chat([  リクエストを送信
012     HumanMessage(content="おいしいステーキの焼き方を教えて")
013 ])
014
```

入力が完了したら以下のコマンドで実行します。

```
python3 chat_model_streaming.py
```

するとこれまでと異なり、以下のように逐次表示されることが確認できるかと思います。

```
以上がおいしいステーキの焼き方です。焼き加減や調味料は好みによって異なるため、自分の好みに合わせてアレンジしてみてください。おいし いステーキの焼き方を教えます。

1. ステーキを室温に戻す: ステーキを冷蔵庫から出して、室温に戻します。これにより、ステーキが均一に焼けます。

2. ステーキを調味する: ステーキに塩とこしょうを振ります。塩はステーキの旨味を引き出し、こしょうは風味を加えます。必要に応じて、他のスパイスやハーブを追加することもできます。

3. フライパンを熱する: フライパンを中火で加熱します。フライパンが十分に熱くなったら、少量の油を敷きます。

4. ステーキを焼く: ステーキをフライパンに入れます。焼く時間はステーキの厚
```

```
さや焼き加減によって異なりますが、一般的には片面2～3分程度焼きます。焼くときには、ステーキにしっかりと火が通るように押さえつけることが重要です。

5. 裏返す: ステーキを裏返し、もう一度2～3分焼きます。焼き加減は好みに応じて調整してください。レア、ミディアムレア、ミディアム、ウェルダンなど、焼き加減はさまざまな種類があります。

6. 余熱させる: ステーキをフライパンから取り出し、余熱させます。これにより、ステーキの中に閉じ込められた旨味が均等に行き渡ります。

7. カットして提供する: ステーキをカットし、お好みの厚さで提供します。お皿に盛り付けて、お好みのソースや付け合わせと一緒に召し上がれます。

以上がおいしいステーキの焼き方です。焼き加減や調味料は好みによって異なるため、自分の好みに合わせてアレンジしてみてください。
```

今回のコードでは6行目でChatOpenAIを初期化する際にstreamingをTrueに、callbacksにStreamingStdOutCallbackHandlerが設定されていることがわかります。

ChatOpenAIは初期化時に与える引数を変更することで動作を変えられます。streamingはTrueに設定することでAPIの呼び出し完了後に処理するのではなく、APIからの応答が到着するたびに処理を行う逐次処理を行なえます。

callbacksには逐次処理する内容を設定します。ここではStreamingStdOutCallbackHandlerが設定されており、結果をターミナル（標準出力）に出力するように設定しています。

13行目で言語モデルを呼び出し、処理を開始しています。

今回のコードでは実行後に結果を表示するためのprint文は存在しません。これは、StreamingStdOutCallbackHandlerで実行される処理内で結果を逐次表示しているため、print文で実行結果を表示する必要がないためです。

もし、結果を表示したうえでソースコードを扱いたい場合は以下のように変更することで取得可能です。

・chat_model_streaming.py

```
~~~省略~~~
resp = chat([  リクエストを送信
    HumanMessage(content="おいしいステーキの焼き方を教えて")
])
response_text = resp.content
```

Language Modelsのcallback機能とstreaming機能を使ってAPIの呼び出し結果を逐次表示できることを確認しました。

03 Templates - プロンプトの構築を効率化する

#プロンプトの構築／#Template
前のセクションでModel I/Oを使った開発を確認しました。今回は、その次のステップとして、プロンプトの構築を簡単にするTemplatesの機能について詳しく解説します。

プロンプトエンジニアリングによる結果の最適化

言語モデルはテキストという形の入力を受け取ります。このテキスト入力はプロンプトと呼ばれます。

GPT-3.5のような最新の言語モデルは、人間が行うような指示を単純な文章で与えても問題なくタスクを実行できる場合も多いですが、単純な指示では実行することが難しいタスクも多くあります。

しかし、プロンプトを最適化することにより、単純な命令では難しかったタスクをこなすことが可能になったり、得られる結果をよりよいものにしたりできます。このプロンプトを最適化する過程、そしてその結果として得られる改善された成果を「プロンプトエンジニアリング」と呼びます。

プロンプトエンジニアリングの効果は大きく、適切なプロンプトで言語モデルを呼び出すことで、以前は不可能と思われていたような高度なタスクも可能になりつつあります。たとえば、科学論文の要約生成、専門知識を要する文章作成、高度なインタラクションなどが可能になってきています。

Templatesモジュールではこのようなプロンプトエンジニアリングを助け、プロンプトの構築を楽にするための機能を提供しています。

出力例を含んだプロンプトを作成する

Model I/OのTemplateモジュールでは前のセクションの「変数をプロンプトに展開する」で学んだような変数と文字列を組み合わせるだけでなく、プロンプトエンジニアリングを含むプロンプトに関わるさまざまな機能を提供しています。プロンプトエンジニアリングの分野では、効果が高いとされている手法が複数存在しますが、その1つであるFewShotPromptについて紹介します。Few-shot promptとは、言語モデルに例を示しながら目的のタスクを実行させる手法です。

具体的には、まず言語モデルが実行すべきタスクについて簡潔に指示し、次に、そのタスクの入力と出力の例をいくつか示します。すると、言語モデルはその例からタ

スクのパターンを学習し、新しい入力が与えられたときに同様の出力を生成できるようになります。

たとえば、文字をアルファベットの大文字に変換するタスクであれば、次のようにfew-shot promptを作成できます。

```
次の例にならって、小文字で入力された文字列を大文字に変換してください:

入力: hello
出力: HELLO

入力: chatgpt
出力: CHATGPT

入力: example
出力: EXAMPLE

入力: {input}
```

このように、実例を示すことで言語モデルは大文字変換のルールを学習し、新しい入力にも適用できるようになります。

Few-shot promptのメリットは、言語モデルに具体的な例を示すことで、人間がイメージする出力に近い結果を生成させられる点です。また、例を変えることで言語モデルの動作を柔軟に制御できます。

Few-shot promptは、言語モデルを使ったアプリケーション開発で広く利用されているテクニックです。LangChainではこのようなFew-shot Promptを簡単に書くための機能を提供しています。

実際にFew-shot promptをLangChainで実装してみましょう。[ファイル]メニューの[新しいてファイル]から、「model_io_few_shot.py」というファイルを作成し、以下の通りに入力してください。

・**model_io_few_shot.py**

```
001 from langchain.llms import OpenAI
002 from langchain.prompts import FewShotPromptTemplate, PromptTemplate
003
004 examples = [
005     {
006         "input": "LangChainはChatGPT・Large Language Model (LLM)の実利
用をより柔軟に簡易に行うためのツール群です",      入力例
```

```
        "output": "LangChainは、ChatGPT・Large Language Model (LLM)の実利用をより柔軟に、簡易に行うためのツール群です。"  出力例
    }
]

prompt = PromptTemplate(  PromptTemplateの準備
    input_variables=["input", "output"],  inputとoutputを入力変数として設定
    template="入力: {input}\n出力: {output}",  テンプレート
)

few_show_prompt = FewShotPromptTemplate(  FewShotPromptTemplateの準備
    examples=examples,  入力例と出力例を定義
    example_prompt=prompt,  FewShotPromptTemplateにPromptTemplateを渡す
    prefix="以下の句読点の抜けた入力に句読点を追加してください。追加して良い句読点は「、」「。」のみです。他の句読点は追加しないでください。",  指示を追加する
    suffix="入力: {input_string}\n出力:",  出力例の入力変数を定義
    input_variables=["input_string"],  FewShotPromptTemplateの入力変数を設定
)
llm = OpenAI()
formatted_prompt = few_show_prompt.format(  FewShotPromptTemplateを使ってプロンプトを作成
    input_string="私はさまざまな機能がモジュールとして提供されているLangChainを使ってアプリケーションを開発しています"
)
result = llm.predict(formatted_prompt)
print("formatted_prompt: ", formatted_prompt)
print("result: ", result)
```

次にPythonで上記コードを実行します。

```
python3 model_io_few_shot.py
```

すると、以下のような結果が確認できます。

```
formatted_prompt:  以下に句読点の抜けた入力に句読点を追加してください。追加して良い句読点は「、」「。」のみです。他の句読点は追加しないでください。
```

```
入力: LangChainはChatGPT・Large Language Model (LLM)の実利用をより柔軟に簡易に行うためのツール群です
出力: LangChainは、ChatGPT・Large Language Model (LLM)の実利用をより柔軟に、簡易に行うためのツール群です。

入力: 私はさまざまな機能がモジュールとして提供されているLangChainを使ってアプリケーションを開発しています
出力:
result:  私は、さまざまな機能がモジュールとして提供されている、LangChainを使ってアプリケーションを開発しています。
```

入力例と出力例に従って句読点の存在しない文章に句読点を追加できました。

コードを詳しく見てみましょう。

4行目では入力例と出力例をリストで持った出力例が設定されています。 ここでは、input、outputをキーにしたオブジェクトの配列を用意しています。 今回設定している例では出力例は1つのみですが、実際は複数の出力例を入力することで目的の結果を得やすくなります。

11行目では5ページと同じように PromptTemplate が設定されています。examplesではinput、outputをキーにしたオブジェクトの配列を渡しているので、PromptTemplateを初期化するさいに input_variables には input、output を渡し、templateには両方を含んだプロンプトを渡しています。

16行目ではFewShotPromptTemplateを準備しています。 このテンプレートは以下を引数として受け取ります。

- examples
  プロンプトに挿入する例をリスト形式で渡します。
- example_prompt
  例を挿入する書式を設定します。PromptTemplateを渡す必要があります。
- prefix
  例を出力するプロンプトの前に置かれるテキストです。今回のコードでは言語モデルへの指示です。
- suffix
  例を出力するプロンプトの後に置かれるテキストです。今回のコードではユーザーからの入力が入ります。
- input_variables
  全体のプロンプトが期待する変数名のリストです。

このように引数で受け取った値を組み合わせてプロンプトを作成しています。

以上で単純な文字列結合で組み込みするよりプログラムで扱いやすい形でプログラムを構築できることが確認できました。

04 Output Parsers - 出力を構造化する

#出力の解析／#OutputParser

6ページでは結果をリスト形式で受け取りました。Output Parsersには、ほかにも便利な機能が用意されています。ここでは実際にどのような機能があるか見てみましょう

結果を日時形式で受け取る

ここでは前の7で作成した「model_io.py」を編集し、結果を日時形式で受け取ってみましょう。［ファイル］メニューの［新規ファイル］から、「datetime_output_parser.py」というファイルを作成し、以下の通りに入力してください。

・datetime_output_parser.py

```
001 from langchain import PromptTemplate
002 from langchain.chat_models import ChatOpenAI
003 from langchain.output_parsers import DatetimeOutputParser  Output ParserであるDatetimeOutputParserをインポート
004 from langchain.schema import HumanMessage
005
006 output_parser = DatetimeOutputParser()  DatetimeOutputParserを初期化
007
008 chat = ChatOpenAI(model="gpt-3.5-turbo", )
009
010 prompt = PromptTemplate.from_template("{product}のリリース日を教えて")  リリース日を聞く
011
012 result = chat(
013     [
014         HumanMessage(content=prompt.format(product="iPhone8")),  iPhone8のリリース日を聞く
015         HumanMessage(content=output_parser.get_format_instructions()),  output_parser.get_format_instructions()を実行し、言語モデルへの指示を追加する
016     ]
017 )
018
019 output = output_parser.parse(result.content)  出力結果を解析して日時形式に
```

```
変換する
020
021 print(output)
022
```

入力が完了したら以下のコマンドで実行します。

```
python3 datetime_output_parser.py
```

すると以下のような表示されます。

```
2020-09-22 00:00:00
```

「発売日は2020年9月22日です」のような文章による返答ではなく、日時形式で返答を受け取ることができました。

主な変更箇所を見てみましょう。6行目では言語モデルからの出力を日時形式へ変換するDatetimeOutputParserをインポートしています。

15行目では受け取りたいのは日時形式なのでプロンプトをリリース日を聞くように入力しています。

以上のようにOutput Parsersを変更することで取得できる構造化データが変更できることを確認できました。

出力形式を自分で定義する

これまでのOutput parsersはLangChainで用意されているものでしたが、Output parsersは自分で作成する形式で受け取ることも可能です。「pydantic_output_parser.py」を新規作成し以下のように入力します。

・pydantic_output_parser.py

```
001 from langchain.chat_models import ChatOpenAI
002 from langchain.output_parsers import PydanticOutputParser
003 from langchain.schema import HumanMessage
004 from pydantic import BaseModel, Field, validator
005
006
```

```
    chat = ChatOpenAI()

class Smartphone(BaseModel):  Pydanticのモデルを定義する
    release_date: str = Field(description="スマートフォンの発売日")
Fieldを使って説明を追加する
    screen_inches: float = Field(description="スマートフォンの画面サイ
ズ(インチ)")
    os_installed: str = Field(description="スマートフォンにインストー
ルされているOS")
    model_name: str = Field(description="スマートフォンのモデル名")

    @validator("screen_inches")  validatorを使って値を検証する
    def validate_screen_inches(cls, field):  validatorの引数には、検証する
フィールドと値が渡される
        if field <= 0:  screen_inchesが0以下の場合はエラーを返す
            raise ValueError("Screen inches must be a positive
number")
        return field

parser = PydanticOutputParser(pydantic_object=Smartphone)
PydanticOutputParserをSmartPhoneモデルで初期化する

result = chat([  チャットモデルにHumanMessageを渡して、文章を生成する
    HumanMessage(content="Androidでリリースしたスマートフォンを1個挙げ
て"),
    HumanMessage(content=parser.get_format_instructions())
])

parsed_result = parser.parse(result.content)  PydanticOutputParserを使っ
て、文章をパースする

print(f"モデル名: {parsed_result.model_name}")
```

ここではPydanticOutputParserを使ってOutput Parsersを作成しています。

PydanticOutputParserは、言語モデルの出力をPydanticモデルに基づいてパースするための便利なツールです。PydanticはPythonでデータ検証を行うライブラリで、型ヒントを使ってデータモデルを定義し、そのモデルに基づいてデータの解析と検証を行います。

PydanticOutputParserを使うメリットは以下の通りです。

- 任意のデータ構造を表現できるPydanticモデルを使ってパースルールを柔軟に定

義できる
- モデルの検証機能を活用して、パースしたデータの整合性を保証できる
- 開発者がPydanticに明示的に定義したデータ構造に解析結果を合わせることができる
- パース結果をPythonオブジェクトとして簡単に取得でき、後続の処理で活用できる

上記コードのSmartphoneクラスは、PydanticのBaseModelを継承したクラスで、スマートフォンの情報を表現するデータモデルです。このモデルは、スマートフォンの発売日（release_date）、画面サイズ（screen_inches）、インストールされているOS（os_installed）、モデル名（model_name）といった情報を持ちます。これらは、型ヒントを使用して定義され、さらにFieldを使ってそれぞれのフィールドの説明を追加しています。

そして、Pydanticのvalidatorを使ってscreen_inchesの値が0より大きいことを確認する検証処理を追加しています。これにより、データをパースする際にscreen_inchesの値が0以下であればエラーが発生します。

21行目ではPydanticOutputParserの初期化し、そのpydantic_objectパラメータに9行目で定義したSmartphoneクラスを渡しています。これにより、チャットモデルからの出力をSmartphoneモデルに基づいて解析できます。

解析は28行目のparser.parse(result.content)で行われ、チャットモデルからの出力（result.content）をSmartphoneモデルに基づいて解析します。結果はparsed_resultに格納され、その各フィールド（model_name、screen_inches、os_installed、release_date）にアクセスすることで、パースした結果を取得できます。

実際に以下コマンドで上記のソースコードを実行してみましょう。

```
python3 pydantic_output_parser.py
```

すると以下のような出力を確認できます。

```
モデル名: Samsung Galaxy S22
画面サイズ: 6.7インチ
OS: Android 12
スマートフォンの発売日: 2022-01-01
```

この出力はチャットモデルから生成されたメッセージをSmartphoneモデルに基づいて解析した結果です。スマートフォンのモデル名、画面サイズ、インストールされているOS、発売日といった情報が適切に取得できています。

このように、PydanticOutputParserは特定の情報を持ったテキストを解析する際に役に立ちます。特に、一定のフォーマットを持ったテキストを解析する必要がある場合や、特定の情報を抽出したい場合に便利です。

たとえば、商品情報を持ったテキストを解析して各商品の詳細情報を取得したり、天気予報のテキストから特定の日の天気を抽出したりすることが可能です。さらに、Pydanticの検証機能を使えば、解析したデータの正確性も確保できます。

誤った結果が返されたときに修正を指示できるようにする

今まで紹介したOutput Parsersでは出力形式の指示をする処理と解析をする処理が存在し、言語モデルが出力への指示にきちんと答えらているという前提でした。

ですが、言語モデルは従来の手続き型プログラミングと異なり、必ず指示を守れるとは限りません。

たとえば、8ページの「リスト形式で結果を受け取る」でのコードだと言語モデルが["iPhone", "Macbook", "iPad"]のような形式で結果を出力することで解析が可能になり、for文で結果を1つずつ取り出せるようになっていました。しかし、必要ない文章や、形式が若干異なる結果を返すことがあります。そのような結果が返されると今まで紹介したコードでは解析する処理（parser.parse()）の行でエラーが起きてしまいます。

実際のアプリケーション開発ではこのようなエラーが発生するのは避けるべきですが、このような問題を解決するためのOutput Parsersも用意されています。先ほど作成したpydantic_output_parser.pyをもとに実際にコードを書いて動きを見ていきましょう。

・pydantic_output_parser.py

```
001 from langchain.chat_models import ChatOpenAI
002 from langchain.output_parsers import OutputFixingParser
OutputFixingParserを追加
003 from langchain.output_parsers import PydanticOutputParser
004 from langchain.schema import HumanMessage
005 from pydantic import BaseModel, Field, validator
006
```

```
007 chat = ChatOpenAI()
008
009 class Smartphone(BaseModel):
010     release_date: str = Field(description="スマートフォンの発売日")
011     screen_inches: float = Field(description="スマートフォンの画面サイ
ズ(インチ)")
012     os_installed: str = Field(description="スマートフォンにインストー
ルされているOS")
013     model_name: str = Field(description="スマートフォンのモデル名")
014
015     @validator("screen_inches")
016     def validate_screen_inches(cls, field):
017         if field <= 0:
018             raise ValueError("Screen inches must be a positive
number")
019         return field
020
021
022 parser = OutputFixingParser.from_llm(      OutputFixingParserを使用するように
書き換え
023     parser=PydanticOutputParser(pydantic_object=Smartphone),      parser
を設定
024     llm=chat      修正に使用する言語モデルを設定
025 )
026
027 result = chat([HumanMessage(content="Androidでリリースしたスマートフォ
ンを1個挙げて"), HumanMessage(content=parser.get_format_instructions())])
028
029 parsed_result = parser.parse(result.content)
030
031 print(f"モデル名: {parsed_result.model_name}")
032 print(f"画面サイズ: {parsed_result.screen_inches}インチ")
033 print(f"OS: {parsed_result.os_installed}")
034 print(f"スマートフォンの発売日: {parsed_result.release_date}")
035
```

コードの変更箇所だけ見ていきましょう。2行目ではOutputFixingParserをインポートしています。OutputFixingParserは先ほど説明した誤った結果を出力したら再実行するためのOutput Parsersです。

22行目でOutputFixingParserを初期化しています。OutputFixingParserはOutput Parsersのリトライをするための Output Parsersなので、23行目のように

リトライする対象のOutput Parsersを入力する必要があります。

24行目はリトライするために使用するLanguage Modelsを設定します。

あとは、同じように実行することで失敗したときにのみ再実行が行われるようになります。