生成AIのAPIを用いたコーディングができるようになる

本日の学習の流れ

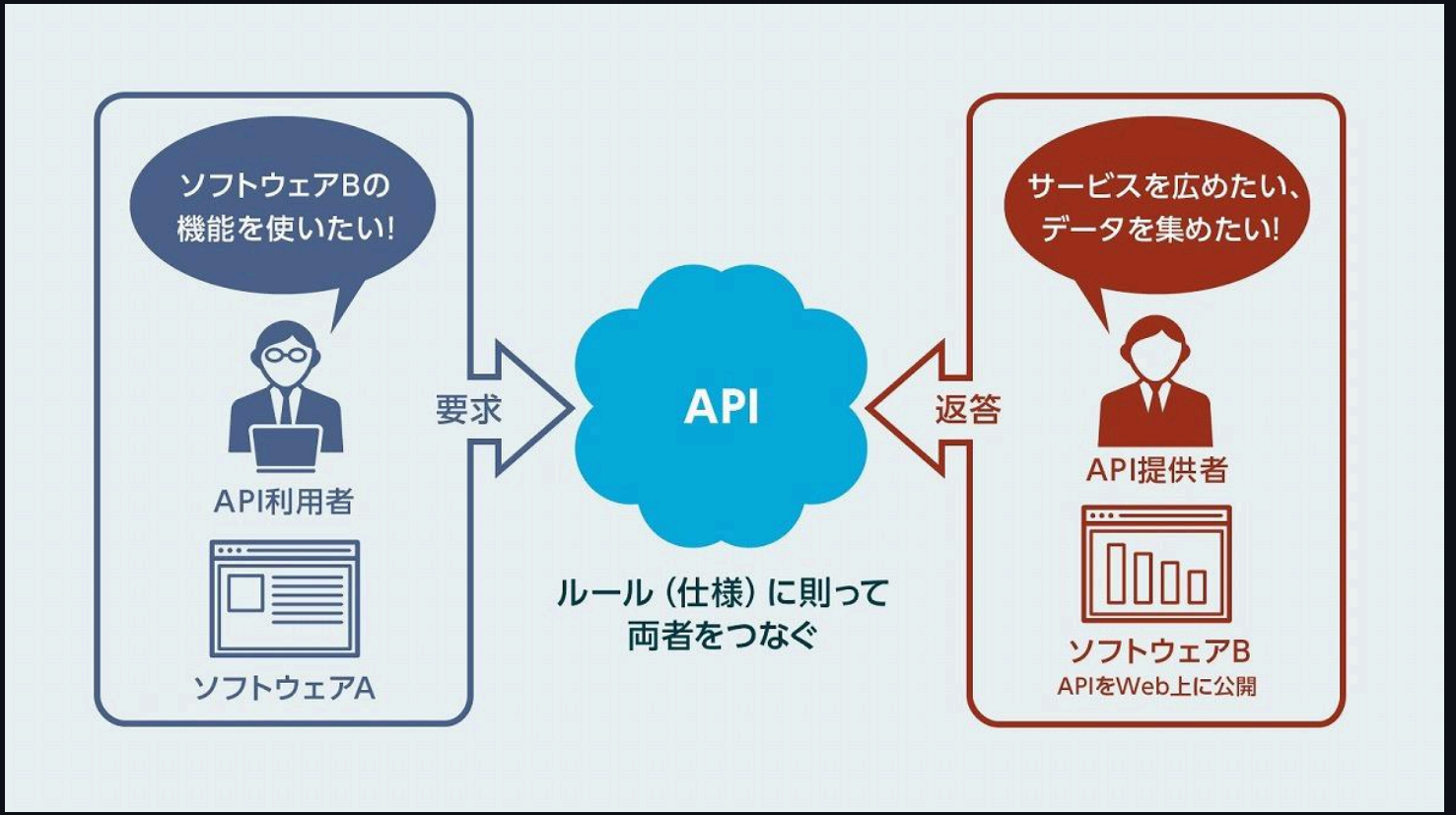
1. **基礎知識:** API、環境変数、パッケージ管理
2. **APIの理解:** 認証、リクエスト/レスポンス
3. **実践:** Gemini APIを使ったコーディング
4. **応用:** パラメータ調整、会話履歴管理

事前知識1: APIとは何か?

API (Application Programming Interface)

APIとは?

APIは、異なるソフトウェア間で情報を交換するためのインターフェースです



- プログラム同士が会話するための「約束事」

身近なAPI:

```
天気予報API:  「東京の天気は?」  →  「晴れ、25度」
翻訳API:  「こんにちは」→  「Hello」
地図API:  「住所」  →  「緯度経度」
生成AI  API:  「俳句を作って」  →  「春の風  花びら舞う  静かな庭」
```

事前知識1: APIを使う利点

なぜAPIを使うのか?

✓ 自分で複雑な機能を実装しなくて済む
✓ 専門家が作った高品質なサービスを利用できる
✓ 最新の技術をすぐに使える

生成AI APIの例:

- 自分でAIモデルを訓練 → 数千万円、数ヶ月
- APIを使う → 数円〜数百円、数分で実装可能

事前知識2: 環境変数とは?

環境変数: プログラムが動作する環境で設定される変数

```
# 環境変数の設定（Linux/Mac）
export GEMINI_API_KEY="your-api-key-here"

# 環境変数の設定（Windows）
set GEMINI_API_KEY=your-api-key-here
```

なぜ環境変数を使う?

- APIキーをコードに直接書かない（セキュリティ）
- 環境ごとに異なる設定を使い分け可能
- Gitなどで誤って公開するリスクを回避

事前知識3: Pythonで環境変数を取得

```python
import os

# 環境変数を取得
api_key = os.environ.get("GEMINI_API_KEY")

# 環境変数が設定されていない場合
if api_key is None:
    print("APIキーが設定されていません")
```

`os.environ.get()`の特徴:

- 存在しない場合は`None`を返す

事前知識4: パッケージ管理

Pythonのパッケージとは?

- 他の開発者が作った再利用可能なコード
- `pip` を使ってインストール

```
# パッケージのインストール
pip install google-genai

# requirements.txtからまとめてインストール
pip install -r requirements.txt
```

requirements.txt:

```
google-genai
```

事前知識5: HTTPリクエストとレスポンス

APIの基本的な流れ:

```
クライアント → [リクエスト] → サーバー
クライアント ← [レスポンス] ← サーバー
```

リクエストに含まれるもの:

- APIキー（認証情報）
- プロンプト（入力データ）
- パラメータ（設定）

レスポンスに含まれるもの:

- 生成されたテキスト
- メタデータ（トークン数など）

事前知識6: トークンとは?

トークン: テキストを分割した単位

```
"こんにちは世界" → ["こんにちは", "世界"] (2トークン)
"Hello World" → ["Hello", " World"] (2トークン)
```

なぜ重要?

- API利用料金はトークン数で計算される
- モデルの入力/出力には上限がある
- 効率的なプロンプト設計に必要

日本語は英語より多くのトークンを消費する傾向

環境準備

必要なパッケージ

```
# requirements.txt
google-genai
```

```
pip install google-genai
```

APIキーの取得

1. Google AI Studio (https://aistudio.google.com/) にアクセス
2. Googleアカウントでログイン
3. 「Get API key」ボタンをクリック
4. プロジェクトを選択または新規作成
5. APIキーをコピーして保存

重要: APIキーは秘密情報として扱い、公開しないこと

基本的なコード構成

```
import os
from google import genai
from google.genai import types
```

主要な要素:

- `genai.Client`：APIとやり取りするクライアント
- `types.Content`：プロンプトと応答を表現
- `types.Part`：メッセージの各部分を構成

Sample 1: 最小構成

code/3/sample_1.py

```
# APIキーを環境変数から取得
api_key = os.environ.get("GEMINI_API_KEY")

# クライアントの初期化
client = genai.Client(api_key=api_key)

# モデルの指定
model = "gemini-flash-lite-latest"
```

ポイント:

- セキュリティのため環境変数からAPIキーを取得
- Flashモデルは高速で軽量

Sample 1: プロンプトの構築

```
contents = [
    types.Content(
        role="user",
        parts=[
            types.Part.from_text(text="俳句を作ってください。"),
        ],
    ),
]
```

構造:

- `Content` : 会話の1つのメッセージ
- `role` : メッセージの送信者（"user" または "model"）
- `parts` : メッセージの内容（テキスト、画像など）

Sample 1: APIの呼び出し

```
generate_content_config = types.GenerateContentConfig()

response = client.models.generate_content(
    model=model,
    contents=contents,
    config=generate_content_config,
)

print(response.text)
```

Sample 1: トークン数の確認

```
print("プロンプトのトークン数:")
print(response.usage_metadata.prompt_token_count)

print("生成されたコンテンツのトークン数:")
print(response.usage_metadata.candidates_token_count)
```

なぜ重要?

- APIの使用量の把握
- コスト管理
- 制限の確認

Sample 2: パラメータの調整

code/3/sample_2.py

```
generate_content_config = types.GenerateContentConfig(
    temperature=1.0,
    top_p=1.0,
)
```

主要パラメータ:

- `temperature` (0.0-2.0): 創造性の制御
  - 0に近い → 決定的・一貫性重視
  - 2に近い → 創造的・ランダム
- `top_p` (0.0-1.0): サンプリング確率の制御

Sample 3: Thinking機能

code/3/sample_3.py

```
generate_content_config = types.GenerateContentConfig(
    thinking_config=types.ThinkingConfig(
        thinking_budget=-1,
    ),
)
```

Thinking機能とは:

- モデルが内部的に思考プロセスを生成
- `thinking_budget=-1`: 無制限に思考可能
- 思考した分のトークン数の追跡: `thoughts_token_count`

Sample 4: モデルの変更

`code/3/sample_4.py`

```
model = "gemini-2.5-pro"
```

モデルの選択:

- `gemini-flash-lite-latest`：高速・軽量
- `gemini-2.5-pro`：高性能・高品質
- 用途に応じて使い分ける

Sample 5: 会話履歴なし

`code/3/sample_5.py`

```
result1 = generate("私は田中と言います。覚えておいてください。")
result2 = generate("私の名前を教えてください。")
```

問題点:

- 各リクエストが独立している
- 前の会話内容を覚えていない
- 名前を尋ねても答えられない

Sample 6: 会話履歴の管理

code/3/sample_6.py

```
conversation_history = []

def generate(prompt: str) -> str:
    # ユーザーのメッセージを履歴に追加
    conversation_history.append(
        types.Content(role="user", parts=[...])
    )

    # 履歴全体を送信
    response = client.models.generate_content(
        model=model,
        contents=conversation_history,
        config=generate_content_config,
    )

    # AIの応答を履歴に追加
```

Sample 6: 会話履歴の効果

履歴あり:

```
User: 私は田中です
AI: 了解しました、田中さん。
User: 私の名前を教えてください。
AI: 田中さんですね。
```

履歴の重要性:

- 文脈を保持
- 自然な会話が可能
- チャットボット等に必須

まとめ

✓ Google Gemini APIの基本的な使い方を学習
✓ プロンプトの構築方法を理解
✓ パラメータによる出力制御
✓ 会話履歴管理の重要性