ChatGPT

OpenAI Responses API チュートリアル (Python開発者向け)

OpenAIの新しい**Responses API**をPythonから活用する方法を、6つのステップに分けて解説します。各コード例では環境変数からAPIキーを読み込む方式を統一して使用します。そのため、事前に以下の準備をしてください：

- `.env` ファイルに `OPENAI_API_KEY` を設定しておく（例: `OPENAI_API_KEY="sk-..."`）。
- Pythonライブラリ `python-dotenv` をインストール済みであること（`pip install python-dotenv`）。
- OpenAIのPythonライブラリは**最新バージョン**にアップデートしておいてください（古いバージョンでは `openai.responses` モジュールが存在せずエラーになります）。

> **Note:** 以下のコード例では、毎回冒頭で `.env` ファイルを読み込み、環境変数からAPIキーを設定しています。同じ処理を各スクリプトの先頭に追加することで、APIキーをハードコーディングせずに安全に管理できます。

それでは、基本的なチャット応答から始め、ファイル検索による高度なQA、ウェブ検索による最新情報の取得、画像の入力・生成（マルチモーダル対応）、過去の応答IDを用いた会話継続、そしてモデルのファインチューニングまで、順に見ていきましょう。

1. Responses APIの基本（通常出力・ストリーミング）

まずは、ChatGPTのような**チャット応答**を得る基本的な使い方です。システムメッセージでAIの振る舞い（ここでは「フレンドリーなPythonの先生」）を指示し、ユーザーメッセージで質問を与えます。Responses APIでは**メッセージのリスト**をそのまま `input` パラメータに渡せるため、ChatCompletion APIと同様の形式で複数メッセージを扱うことができます（もちろん単純な文字列を渡すことも可能です）。以下はシステムとユーザーのメッセージを含む基本的な例です。

python コピーする

```python
from dotenv import load_dotenv
import os
import openai

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# システムメッセージとユーザーメッセージを定義
messages = [
    {"role": "system", "content": "あなたはフレンドリーなPythonの先生です。"},
    {"role": "user", "content": "Pythonで現在の日時を表示するにはどうすれば良いですか？
]

# Responses API を使って応答を取得
response = openai.responses.create(
```

```
    model="gpt-4o",            # 利用するモデル（GPT-4系列のモデルを指定）
    input=messages             # メッセージのリストをそのまま input 引数に渡す
)

# 応答メッセージを出力
print(response.output_text)
```

上記コードでは、`messages` リストにシステム役割とユーザーからの質問を設定し、`openai.responses.create` に渡しています。モデルには `gpt-4o`（GPT-4 系列のモデル）を指定していますが、OpenAIアカウントで使用可能なモデルであれば自由に選択できます。【GPT-3.5系モデルでも基本機能は試せますが、一部高度な機能（ツール使用や画像入力など）はGPT-4系列でのみサポートされています。】`input` にリストを渡すことで複数のメッセージ（ロール付き）を一度に提供でき、AIはそれらをすべて考慮した上で回答を生成します。

実行結果: 上のコードを実行すると、AIからの回答テキストが `response.output_text` に格納されています。例えば、質問に対して「`datetime` モジュールを使って現在日時を取得する方法」がフレンドリーな口調で説明されるでしょう。

ストリーミング応答の取得

通常の呼び出しではAIの応答全文が生成されてから返ってきますが、`stream=True` オプションを指定すると、応答が生成され次第逐次データを受け取ることができます。これにより、長い回答でもリアルタイムに内容を出力したり、ユーザーに逐次表示したりすることが可能です。ストリーミングモードを利用する場合、レスポンスはイベントストリーム（イテレータ）として返されるため、ループで処理します。

python コピーする

```
from dotenv import load_dotenv
import os
import openai

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# 単純なユーザーからの質問（文字列でも可）
user_input = "PythonでFizzBuzz問題を解くコードを書いてください。"

# ストリーミングモードでAPI呼び出し
stream = openai.responses.create(
    model="gpt-4o",
    input=user_input,
    stream=True
)

# 部分応答を順次受け取って表示
for event in stream:
    # event.delta に増分テキストが含まれる
    if hasattr(event, "delta") and event.delta:
        print(event.delta, end="", flush=True)
```

`stream=True` を指定した場合、戻り値の `stream` は逐次到着するイベントのイテレータとなります。各イベントオブジェクトの `delta` 属性に追加のテキストが含まれているので、上記のように `for` ループでそれを取り出して出力します。【ストリーミング利用時は、単純に `response = openai.responses.create(..., stream=True)` としても最終結果は得られません。必ずループでイベントを処理する実装が必要です。イベント構造を無視して `print(response)` などとしないよう注意してください。】

> **Note:** Responses API機能を利用するには、OpenAIのPythonライブラリを最新にしておく必要があります。古いバージョンでは `responses` モジュールが存在せずエラーになりますので、pipでアップデートしてから実行しましょう。

2. ベクトルストアを用いたファイル検索（ドキュメントQA）

続いて、**社内ドキュメントなどのデータをAIに読み込ませて回答させる**方法です。OpenAI APIでは、事前にアップロードしたドキュメントをベクトル（埋め込み）化してインデックスを作成し、質問内容にマッチするテキスト片を自動検索して回答に組み込むことができます。ユースケース例として、社内Wikiやマニュアルからの QA、自社製品のFAQ回答、契約書の内容確認などが挙げられます。

Responses APIでは、この機能を「**ファイル検索 (`file_search`)」ツール** として組み込むことができます。あらかじめ**検索対象のファイルをOpenAIのサーバーにアップロードし、ベクトルストア（Vector Store）に登録**しておく必要があります。以下のコード例（ `step2_vector_search.py` ）では、テキストやPDFファイルを想定した手順を示します。

python コピーする

```python
from dotenv import load_dotenv
import os
import openai

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# 1. アップロードするファイルを準備（テキストまたはPDF）
file_path = "company_faq.pdf"  # ここに自分のファイルパスを指定
with open(file_path, "rb") as f:
    file_data = f.read()

# 2. ファイルをOpenAIにアップロード（目的を "assistants" に指定）
upload_response = openai.File.create(file=file_data, purpose="assistants")
file_id = upload_response.id
print(f"Uploaded file ID: {file_id}")

# 3. 新規ベクトルストアの作成
vector_store = openai.vector_stores.create(name="knowledge_base")
vector_store_id = vector_store.id
print(f"Vector store ID: {vector_store_id}")

# アップロードしたファイルをベクトルストアに追加（インデックス登録）
openai.vector_stores.files.create(vector_store_id=vector_store_id, file_id=file_id)
print("File added to vector store. (インデックス作成には少し時間がかかります)")

# 4. ファイル検索ツールを使って質問を投げかける
query = "社内向けドキュメントのパスワード再設定手順は？"
response = openai.responses.create(
    model="gpt-4o",
```

```
    input=query,
    tools=[{"type": "file_search", "vector_store_ids": [vector_store_id]}]
)
print(response.output_text)
```

コードの解説:

1. **ファイルのアップロード:** まず `openai.File.create` でローカルファイルをアップロードしています。`purpose="assistants"` とすることで、このファイルがアシスタント（Responses API）の検索用データとして扱われます【このpurpose指定によりアップロードされたファイルがベクトルストアに登録可能となります】。戻り値から `file_id`（ファイルID）が取得できます。
   ※ファイルパスの代わりにURLを指定し、Pythonの `requests` 等でダウンロードしてアップロードすることも可能です。
2. **ベクトルストアの作成:** `openai.vector_stores.create` で新たなベクトルストア（例では `"knowledge_base"` という名前）を生成します。ベクトルストアはアップロードした複数ファイルからなる**検索用インデックス**であり、`vector_store_id` を控えておきます。
3. **ファイルをベクトルストアに追加:** `openai.vector_stores.files.create` を使い、先ほどの `file_id` をベクトルストアに登録します。この時点でOpenAI側でファイル内容のEmbedding（ベクトル化）作成とインデックス化が行われます（完了まで数十秒程度かかる場合があります）。登録状況は `openai.vector_stores.files.list` で確認することもできます。
4. **ファイル検索ツールで質問:** 準備が整ったら、`openai.responses.create` で実際に質問を投げかけます。ここで `tools` 引数に `{"type": "file_search", "vector_store_ids": [vector_store_id]}` を指定するのがポイントです。この設定により、AIは回答を生成する際に指定したベクトルストア内のファイルを検索し、関連情報を取得できます。上の例では `query` として社内ドキュメントに関する質問を与えています。
5. **結果の表示:** 最後に `response.output_text` を表示すると、AIがファイルから見つけた情報に基づいて回答を返してくれます。例えば「パスワードリセット手順を教えて」という問いに対し、アップロードしたFAQ PDFから該当する手順を抜き出して説明してくれる、といった回答が得られるでしょう。

ユースケース例: 社内FAQやマニュアルを読み込ませておけば、新人社員からの質問にAIが即座に回答したり、過去のナレッジベースを検索してベテラン社員の知見を引き出したりといった使い方ができます。複数のファイルを一つのベクトルストアに追加しておけば、横断的な検索も可能です。

> **注意点:**
> **・アップロード可能なファイル形式:** 現在テキスト（.txt）やPDF（.pdf）などがサポートされています。PDFの場合はテキスト抽出が自動で行われます。ただし画像のみのPDFなど、文字情報が無いものは正しく抽出できない可能性があります。
> **・ファイルサイズとトークン制限:** 一度にアップロードできるファイルサイズや、検索時に参照できるトークン数には上限があります。巨大なドキュメントは適切に分割する、古い情報は適宜除外するなどして運用してください。

3. Web検索ツールの利用（最新情報へのアクセス）

AIがユーザーの質問に答える際に、**自動でインターネット検索を行い最新の情報を取得**することもできます。ユースケース例として、ニュースの要約【今日の主要ニュースを教えて】、商品のリアルタイム価格調査、天気や交通情報の案内など、**時事性のある情報**への回答が挙げられます。

Responses APIでウェブ検索機能を使うには、API呼び出し時に `tools` パラメータで `{"type": "web_search_preview"}` を指定します。これによりモデルが必要に応じて自動的に検索クエリを構築し、関連するWebページから情報を取得して最終回答を作成します。【質問内容からモデル自身が「検索すべきかどうか」を判断し、必要であれば検索を実行する仕組みです。】下記の例では、今日のAI業界のニュースを尋ねており、モデルは内部で検索を行って回答を生成します。

```python
from dotenv import load_dotenv
import os
import openai

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# 例: 最新のテクノロジーニュースに関する質問
query = "今日のAI業界の重要なニュースを3つ教えてください。"

response = openai.responses.create(
    model="gpt-4o",
    input=query,
    tools=[{"type": "web_search_preview"}],   # Web検索ツールを有効化
    tool_choice="required"                    # ツール使用を必須に設定（後述）
)

print("AIの回答:")
print(response.output_text)
```

上記のコードでは、`tools` にWeb検索ツールを指定しただけで、AIは回答生成の途中で必要に応じてインターネット検索を行います。例えばこの質問では「今日のAI業界の重要なニュース」という**最新情報**が求められているため、モデルは自律的に検索クエリを作成し、ニュースサイト等から情報を取得して、その結果に基づいた回答（ニュース概要）を返します。実行すると、AIから今日のAI関連ニュース3件についての要約が得られるでしょう（出典となるニュース記事のタイトルやURLが回答中に示される場合もあります）。

tool_choiceパラメータによるツール使用の制御

デフォルトでは、モデルは質問内容に応じてツールを使うかどうか**自動**判断します。しかし場合によっては、開発者側で「必ずツールを使ってほしい」あるいは「特定のツールは使ってほしくない」と指定したいこともあります。そこで `tool_choice` というパラメータでモデルのツール使用方針を制御できます。

- `"auto"`（デフォルト）：モデルがツールを使うかどうか自律判断します。必要なら使いますが、不要と判断すれば使いません。
- `"required"`：**必ず**何らかのツールを使用します（指定した複数ツールがあればその中からモデルが選択します）。今回のように1つだけツールを与えた場合はそのツールを確実に使って回答を生成します。
- `"none"`：ツールを一切使わないモードです（たとえ `tools` を与えていても無視します）。

※さらに高度な指定として、`tool_choice` に特定のツール名（例えば `"web_search_preview"` など）を与えると、「与えたツール群の中からこのツールだけを使え」と厳密に指示することも可能です。ただし一般的な用途では上記の `"required"` まで指定できれば十分でしょう。

4. 画像入力と画像生成（マルチモーダル対応）

このステップでは、画像を扱う入出力に対応する方法を学びます。OpenAIのAPIでは、**画像生成**（テキストから画像を作る）と**画像内容の理解**（画像を入力してその内容について質問する）の両方が可能です。前者はいわゆるDALL·Eによる画像生成機能で、後者はGPT-4oなどマルチモーダル対応モデルによる画像解析です。

できることの概要: ユースケース例として、デザイン案の自動生成「◯◯のイラストを描いて」→AIが画像生成）、画像の内容説明（写真を入力して「この画像に写っている建物は何ですか？」と質問）、画像診断（医療画像を与えて所見を聞く）などが挙げられます。Responses APIを通じてこれらをプログラムで実現できます。

4-1. 画像を入力して質問する（画像の解析）

まず、**画像を入力として与え、その内容についてAIに質問する方法**です。GPT-4oなど一部のモデルは画像をインプットとして解析し、質問に答えることができます。Responses APIでは、**メッセージの一種として画像を渡す**仕様になっています。

ポイント: ユーザーメッセージの `content` に画像を含める場合、少し特殊な構造で渡します。具体的には以下のように、`content` にリスト形式で `{"type": "input_image", "image_url": "画像のURL"}` オブジェクトを入れます。この `image_url` にはOpenAIのサーバーからアクセス可能な場所にホストされた画像のURLを指定してください（ローカルの画像パスを直接指定することはできません）。

python コピーする

```python
from dotenv import load_dotenv
import os
import openai

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# ユーザーからの質問文と画像URLを準備
question = "この犬の品種は何ですか？"
image_url = "https://upload.wikimedia.org/wikipedia/commons/2/20/Shiba_Inu.jpg"

# 画像を含むユーザーメッセージを構築
messages = [
    {"role": "user", "content": question},
    {"role": "user", "content": [ {"type": "input_image", "image_url": image_url} ]
]

response = openai.responses.create(
    model="gpt-4o",
    input=messages
)

print("AIの回答:")
print(response.output_text)
```

上の例では、柴犬の写真を示す画像URLと「この犬の品種は何ですか？」という質問を組み合わせてモデルに与えています。モデルは画像内容（犬の見た目）を解析し、テキスト質問と合わせて回答を生成します。実行すると、例えば**「この犬は柴犬です」**といった回答が得られるでしょう。

4-2. テキストから画像を生成する（画像生成）

次に、**テキストプロンプトから画像を生成する方法**です。こちらはDALL·E 3モデルによる画像生成で、OpenAI APIでは専用のエンドポイント（Image API）を使用します。2023年以前は `openai.Image.create()` メソッドでDALL·E 2による画像生成を行っていましたが、2025年現在、新たに**GPT-4ベースの画像生成モデル** `gpt-image-1` がAPIで利用可能になっています。このモデルはDALL·E 3に相当する高品質な画像生成を行いますが、使い方は従来と同様で、テキストで記述したプロンプトを与えると画像の一時URLが返されます。コード例（ `step4_image_generation.py` ）を示します。

python コピーする

```python
from dotenv import load_dotenv
import os
import openai
import requests

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

prompt = "柴犬が宇宙服を着て月面を歩いているイラスト"

response = openai.Image.create(
    prompt=prompt,
    n=1,            # 生成する画像の数
    size="512x512"  # 画像サイズ（256x256, 512x512, 1024x1024 など）
)

image_url = response["data"][0]["url"]
print(f"Generated image URL: {image_url}")

# オプション: 画像をダウンロードしてファイル保存する場合
img_data = requests.get(image_url).content
with open("output.png", "wb") as f:
    f.write(img_data)
print("画像を output.png に保存しました。")
```

コードの解説:

- `openai.Image.create` メソッドに対し、生成したい画像の内容をテキストで `prompt` として渡します。例えば「柴犬が宇宙服を着て月面を歩いているイラスト」というプロンプトから、それに対応する画像を作成してくれます。
- `n` は生成する画像の枚数です。必要に応じて複数枚生成も可能ですが、**1枚ごとに料金がかかる**点に注意してください。
- `size` で画像の解像度（ピクセル寸法）を指定できます。一般的には `512x512` がバランス良いです（ `1024x1024` も指定可能ですが、生成に時間がかかったりコストが増えたりします）。
- APIのレスポンスには、生成された画像の**一時URL**が含まれています。`response["data"][0]["url"]` でそのURLを取得可能です。このURLは一定時間（数時間程度）有効なので、その間に画像を取得してください。必要に応じて、上記コードのように `requests` でダウンロードしてローカルファイル（例では `output.png` ）に保存できます。

上記の例では1枚だけ生成していますが、`n` を増やせば複数の異なるバリエーション画像を一度に得ることもできます。`gpt-image-1` （DALL·E 3相当）の登場により、生成される画像の質が向上しており、非常に詳細で実用的なイラストや写真を得ることができます。

5. previous_response_id を使ったスレッド継続（会話の文脈保持）

最後に、**過去の応答IDを利用して会話を継続する方法**を紹介します。一問一答ではなく、直前までの対話の文脈を踏まえた連続的な会話を行うことで、ChatGPTのように前の発言内容を踏まえた応答を得ることができます。例えばユーザーが「この前教えてもらった方法で試したけどうまくいかない」と言った場合、AIが「前回提案した方法」という文脈を理解した上で追加のアドバイスを行う、といったことが可能になります。また、複数ユーザーを相手にするチャットボットサービスでは**各ユーザーごとに別々の会話スレッド**を維持することが重要ですが、この機能を使えばサーバー側でユーザーごとの会話履歴IDを管理できます。

Responses APIで会話を継続するには、**直前の応答のID**を次のリクエストに渡します。具体的には `openai.responses.create` 呼び出しの際に、`previous_response_id` パラメータに直前に受け取った `response.id` をセットします。これにより、モデル側で「以前の会話」が関連付けられた状態で新たな応答が生成されます。

以下に、簡単な連続対話のコード例（ `step5_conversation_thread.py` ）を示します。

python コピーする

```python
from dotenv import load_dotenv
import os
import openai

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# ユーザーとの最初のやり取り
message1 = "フランスの首都はどこですか？"
response1 = openai.responses.create(
    model="gpt-4o",
    input=message1
)
print(f"Q: {message1}")
print(f"A: {response1.output_text}")  # 例: "フランスの首都はパリです。"

# previous_response_idを使って次の質問を継続
message2 = "そこには何人の人が住んでいますか？"  # 前の回答で出た「そこ（パリ）」に言及し
response2 = openai.responses.create(
    model="gpt-4o",
    input=message2,
    previous_response_id=response1.id   # 前の応答IDを指定して文脈を維持
)
print(f"Q: {message2}")
print(f"A: {response2.output_text}")  # 例: "パリには約214万人（2020年時点）が住んでい
```

コードの解説:

まず最初の質問 `message1` に対する `response1` を取得しています。この `response1` オブジェクトには `id` プロパティとして**ユニークな応答ID**が割り当てられています。次にユーザーが続けて質問 `message2` をしたとします。このとき `openai.responses.create` を呼ぶ際に、`previous_response_id=response1.id` を含めることで、モデルは直前の会話（Q&A）を踏まえた上で `message2` に答えてくれます。

実際、例では `message2` が「そこには何人の人が住んでいますか？」という曖昧な質問ですが、直前の会話で「そこ（=パリ）」という文脈が確立しているため、AIは「パリの人口」を答えることができます。上のコードを実行すると、1つ目の質問に対する答え「パリです」に続き、2つ目の質問に対して「パリには約〇〇万人が住んでいます」という具合に、**会話が繋がった応答**が得られるはずです。

注意:

- “ `previous_response_id` **の有効期限には注意しましょう。一度取得した応答IDは一定期間有効ですが、極端に古いIDを渡すと履歴が見つからずエラーになる可能性があります。基本的には同一ユーザーとのセッション内で最新のIDを使い、長期間経過した古いスレッドを復元しようとしない方が安全です。**”
- **“独立性の確保: 異なるユーザーの会話履歴IDを誤って使い回すと、他ユーザーの会話内容が漏洩するリスクがあります。必ずユーザー単位で対応するIDを分離・管理してください（例：ユーザーIDごとに最新のresponse.idを保存しておき、それぞれ別々に会話を継続する）。”**

6. モデルのファインチューニング（purpose="fine-tune"）

最後に、**モデルのファインチューニング**（Fine-tuning）によって、自分のデータでモデルを微調整する方法を紹介します。ファインチューニングとは、既存のベースモデルに追加の学習をさせて、出力の傾向やスタイルをカスタマイズすることです。これにより、プロンプトに対して**より望ましい形式で回答**するモデルを作成したり、特定のタスクに特化したモデルを得たりできます。

例えば、自社のカスタマーサポートの会話データでファインチューニングして口調や回答内容をブランド仕様に合わせたり、専門分野のQ&Aペアで微調整して専門用語をより正確に扱えるようにしたり、といった応用が考えられます。【なお、モデルに新たな知識を覚えさせる用途にはファインチューニングは推奨されません。特定ドメインの知識を取り込むには前述のベクトル検索のような手法が適しています。ファインチューニングは主に**応答のスタイルやフォーマットの調整**に有効です。】

OpenAIのAPIでは、テキストCompletionモデルおよび一部のチャットモデルに対してファインチューニング機能が提供されています。ここではGPT-3.5ターボシリーズを例に、ファインチューニングの手順を説明します。

ファインチューニングの手順概要:

1. **トレーニングデータの準備:** まず、モデルに学習させたい**入出力ペア**のデータセットを用意します。チャット形式のモデルをチューニングする場合、各データは以下のようなJSON Lines形式（ `.jsonl` ファイル）になります。

   ```json
   {"messages": [
       {"role": "user", "content": "<ユーザーの発話1>"},
       {"role": "assistant", "content": "<アシスタントの返答1>"}
   ]}
   {"messages": [
       {"role": "user", "content": "<ユーザーの発話2>"},
       {"role": "assistant", "content": "<アシスタントの返答2>"}
   ]}
   ```

   （※システムメッセージや複数ターンを含む学習も可能です。）このように1行ごとに会話例をJSONオブジェクトで記述したテキストファイルを作成します。ここでは仮に `finetune_data.jsonl` というファイル名で準備できたとします。

2. **ファイルのアップロード（purpose="fine-tune"):** 上記のトレーニングデータファイルをOpenAIにアップロードします。

3. **ファインチューニングジョブの作成:** アップロードしたデータを用いてファインチューニングを開始します（ベースとするモデルを指定できます）。
4. **ジョブの完了待ち:** ファインチューニングジョブが完了するまで待機します（数分〜十数分程度かかる場合があります）。完了後、ファインチューニングされた新たなモデルが作成され、そのモデルIDが取得できます。
5. **ファインチューニング済みモデルの利用:** 得られたモデルIDを指定してAPIを呼び出すことで、ファインチューニング後のモデルを利用できます。

それでは、これらの手順をコードで実行してみます。

python コピーする

```python
from dotenv import load_dotenv
import os
import openai

load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# 1. トレーニングデータファイルをアップロード（purpose を "fine-tune" に指定）
training_file_path = "finetune_data.jsonl"  # 準備した学習データファイル
file_response = openai.File.create(
    file=open(training_file_path, "rb"),
    purpose="fine-tune"
)
training_file_id = file_response.id
print(f"Uploaded training file ID: {training_file_id}")

# 2. ファインチューニングジョブの作成（ベースモデルに gpt-3.5-turbo を指定）
fine_tune_response = openai.FineTuningJob.create(
    training_file=training_file_id,
    model="gpt-3.5-turbo"
)
fine_tune_job_id = fine_tune_response.id
print(f"Fine-tuning job started. ID: {fine_tune_job_id}")

# 3. ジョブの完了を待機（ステータスをポーリング）
import time
status = "running"
while status not in ["succeeded", "failed"]:
    job_status = openai.FineTuningJob.retrieve(fine_tune_job_id)
    status = job_status.status
    print(f"Job status: {status}")
    if status == "running":
        time.sleep(10)  # 10秒待って再度ステータス確認

# 4. ジョブ完了後、ファインチューニングされたモデル名を取得
if status == "succeeded":
    fine_tuned_model = job_status.fine_tuned_model
    print(f"Fine-tuned model: {fine_tuned_model}")

    # 5. ファインチューニング済みモデルを使って応答を取得
    test_prompt = "カスタマイズモデルの動作テストです。こんにちは。"
    response = openai.responses.create(
        model=fine_tuned_model,
        input=test_prompt
```

```
    )
    print(response.output_text)
else:
    print("Fine-tuning job failed or did not complete.")
```

コードの解説:

- **ファイルアップロード:** 学習用データファイル（ `.jsonl` ）を `purpose="fine-tune"` としてアップロードします。返り値から `training_file_id` が得られます。
- **ジョブ作成:** `openai.FineTuningJob.create` にトレーニングファイルIDとベースモデル（例: `"gpt-3.5-turbo"` ）を指定してジョブを開始します。返り値から `fine_tune_job_id` （ジョブID）が取得できます。
- **ステータス確認:** ジョブIDを使って `openai.FineTuningJob.retrieve` を呼ぶと、`status` （running/succeeded/failed など）や進行状況を確認できます。上では簡易的にポーリングして完了を待っています（実運用ではWebフックを設定してジョブ完了通知を受け取ることも可能です）。
- **モデル名取得:** ジョブが完了（succeeded）すると、`job_status.fine_tuned_model` に新しく出来上がったモデルの識別子が入ります。この識別子（例えば `"ft:gpt-3.5-turbo-..."` のような文字列）を使って、以後のAPI呼び出し時にモデルを指定します。
- **モデルの利用:** `model` パラメータに上で得たモデル識別子を指定し、`openai.responses.create` を呼ぶことで、ファインチューニング済みモデルが応答します。使い方は通常のモデルと全く同じです。

ファインチューニングしたモデルはベースモデルに比べ、与えたデータセットに即した振る舞いをします。例えば、カスタマーサポートの対話データでチューニングしていれば、その口調や回答パターンが反映された応答が得られるでしょう。**ファインチューニングには追加のコスト**（トレーニング実行コストと、生成時のトークン単価）が発生しますが、うまく活用すればモデル応答の品質向上や一貫性の確保に役立ちます。

> **注意:** ファインチューニングを行う際は、OpenAIの利用規約やコンテンツポリシーを遵守してください。学習データに個人情報や機密情報を含める場合は十分に注意し、必要に応じて匿名化・マスキングを行いましょう。また、ファインチューニングしたモデルも新たな知識を得るわけではないため、最新の事実情報などは引き続きStep2やStep3のような補助手段と組み合わせる必要があります。

おわりに

以上、OpenAIの新しいResponses APIを使って実現できる主要な機能を6つのステップに分けて紹介しました。基本的なチャット応答から始まり、ドキュメント検索による高度なQA、ウェブ検索による最新情報の取得、画像の理解・生成といったマルチモーダル対応、会話の継続、そしてモデルのファインチューニングまで、一通りの使い方を網羅できたかと思います。

このチュートリアルを通じて、Python開発者の皆さんがResponses APIの強力な機能を体験し、自身のアプリケーションに組み込むイメージを掴んでいただけたなら幸いです。例えば、社内ヘルプデスクBOTに組み込んでドキュメント回答や最新ニュース提示ができたり、ユーザーからアップロードされた画像を解析して自動応答するサービスを作れたりと、応用範囲は非常に広いです。

最後に、開発時の注意事項として、OpenAI APIの利用にあたっては**利用規約やプライバシー規制**にも留意してください。社内データをアップロードする際は暗号化やアクセス制限を適切に行い、APIから得た情報の扱いも慎重にする必要があります。また、本記事執筆時点（2025年）の仕様に基づいておりますので、将来的にAPI仕様が変わった場合は公式ドキュメントを参照して最新情報を確認してください。

ぜひ実際にコードを動かしながら、OpenAI Responses APIの可能性を探ってみてください。あなたの作るPythonアプリケーションが、この強力なAPIによってさらに豊かなユーザー体験を提供できることでしょう。
Happy Coding!

参考情報

- **Implementing OpenAI's Responses API: A Beginner's Guide** – Arsturn Tech Blog
  https://www.arsturn.com/blog/a-beginners-guide-to-implementing-openais-responses-api
- **OpenAI Responses API: A Comprehensive Guide** – Tom Odhiambo (Medium)
  https://medium.com/@odhitom09/openai-responses-api-a-comprehensive-guide-ad546132b2ed

---

すべての情報源