3.2 프롬프트 구조 실습

3.2.1 Role 기반 프롬프트

① Role은 LLM의 말투·관점·설명 수준을 결정하는 핵심 요소

LLM은 사용자가 지정한 역할(Role)에 따라 말투, 설명 방식, 전문성 수준이 달라집니다. 이 역할 지정은 `system` 메시지를 통해 이루어지며, 모델에게 "어떤 사람처럼 말하라" 또는 "어떤 상황을 가정하고 답하라"는 규칙을 부여하는 방식입니다.

예를 들어 "당신은 대학 강의 교수입니다."라고 설정하면 보다 전문적이고 체계적인 설명을 제공하고, "초등학생에게 설명하는 선생님입니다."라고 설정하면 쉬운 단어와 비유 중심의 답변을 만들게 됩니다. Role은 출력 품질을 조정하는 가장 기본적이면서도 강력한 도구입니다.

② 전문가 역할을 부여하면 톤과 논리 구조가 안정적으로 유지됨

특정 역할(예: 기획자, 번역가, 분석가, 개발자 등)을 설정하면 LLM은 해당 전문가의 관점과 사고 방식에 맞춘 답변을 제공합니다. 예를 들어 기획자 역할을 주면 문제 정의 → 분석 → 해결 전략이라는 구조적 흐름을 유지하고, 번역가 역할을 주면 자연스럽고 정확한 번역을 출력합니다. 특정 역할을 지속적으로 유지하는 것은 출력 품질의 일관성을 확보하는 핵심 원리입니다.

③ 여러 단계로 구성된 생성 작업에서 Role 설정의 중요성은 더욱 커짐

기획 문서 작성, 보고서 초안 생성, 콘텐츠 제작, 스크립트 작성 등 여러 단계를 거치는 작업에서는 톤과 관점이 중간에 흐트러지기 쉽습니다. Role 설정이 명확하면 모델이 작업 내내 일관된 스타일과 전문성을 유지합니다. 따라서 고급 프롬프트 설계와 AI Agent 구현에서도 Role 지정은 반드시 포함되는 요소입니다.

실습 목표

- 동일한 질문이라도 Role에 따라 출력 방식이 크게 달라지는 것을 비교·확인한다.
- `system` 메시지가 모델의 "말하기 방식"을 어떻게 제어하는지 이해한다.
- 전문가 역할과 일반 역할의 차이를 직접 경험한다.

실습. 역할(role)에 따른 설명 방식 비교

아래 실습은 입력한 주제를 **세 가지 역할(교수, 초등학생 선생님, 마케팅 전문가)** 관점으로 설명해보는 실험입니다.

이 단원의 난이도에 맞게 **가장 명확하게 차이를 체감할 수 있는 단순 구조**로 검증된 코드입니다.

```
from openai import OpenAI
from dotenv import load_dotenv
import os

load_dotenv()
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

topic = input("설명할 주제를 입력하세요: ")

roles = [
    "당신은 대학 강의 교수입니다.",
    "당신은 초등학생에게 설명하는 선생님입니다.",
    "당신은 마케팅 전문가입니다."
]

for r in roles:
    resp = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": r},
            {"role": "user", "content": f"{topic}를 설명하시오."}
        ]
    )

    print("\n========================================")
    print(f"[Role 적용] {r}")
    print("----------------------------------------")
    print(resp.choices[0].message.content)
```

① 역할에 따라 설명 방식이 어떻게 변화하는가?

- 교수 역할 → 개념 정의, 구조화된 설명, 전문 용어 사용

- 초등학생 대상 → 쉬운 단어, 짧은 문장, 비유 중심
- 마케팅 전문가 → 메시지 전달력 강조, 설득 구조 활용

② “같은 질문”임에도 Role만 바뀌어도 출력 품질이 크게 달라진다

이는 LLM이 단순 Q&A 생성기가 아니라 **역할 기반 언어 생성 시스템**이라는 성격을 보여준다.

③ Role 설정은 이후 모든 문서 자동화·에이전트 구성의 기본

특히 멀티 스텝 작업에서 톤과 논리 구조를 유지하는 데 필수이다.

3.2.2 Format 기반 프롬프트

① 출력 형식을 명확히 지정하여 원하는 구조로 정확한 결과를 얻는 방법

LLM은 매우 유연한 언어 생성 능력을 가지고 있지만, 출력 형식을 지정하지 않으면 문단, 목록, 요약, 번호 등 원하는 구조를 벗어나는 답변이 나올 수 있습니다. Format 기반 프롬프트는 모델에게 **출력 형태를 명확히 지시하여, 구조적·일관된 결과를 얻는 방식**입니다. 예를 들어 “3줄 요약”, “표 형식”, “JSON 형식”, “번호 목록”과 같이 요구하면 모델은 해당 형식에 맞춘 결과를 생성합니다. 이 방식은 실제 업무 보고서, 데이터 전처리, API 연동 등에서 필수적으로 사용됩니다.

② 형식을 미리 정하면 품질 관리가 쉬워지고 후처리 자동화가 가능함

Format을 명확하게 지정해두면 모델의 출력이 **예측 가능하고 재현성 높은 구조**로 생성됩니다. 예를 들어 AI Agent 구조에서는 LLM이 일정한 JSON 필드를 유지해야 파서가 안정적으로 작동하며, 자동화된 콘텐츠 생성에서는 표준 형식이 맞지 않으면 전체 워크플로우가 깨질 수 있습니다. 따라서 Format 기반 프롬프트는 단순한 출력 통제가 아니라 **품질 관리(QA)와 자동화 가능성**을 확보하는 핵심 기술입니다.

③ LLM과 외부 시스템을 연결할 때 필수적인 제어 방식

멀티포맷 콘텐츠 생성, 데이터 분석 자동화, 함수 호출(Function Calling), API 기반 에이전트 구성 등에서는 LLM이 특정 형식(JSON, 키-값, 템플릿)을 엄격하게 준수해야 합니다. Format 지정을 하지 않은 답변은 자연스럽지만 일관성이 부족하여 오류를 유발할 수 있습니다. 반면 Format을 명확히 제시하면 LLM 출력이 외부 시스템에서 쉽게 읽혀 **인공지능–시스템 간 통신 품질이 크게 향상**됩니다.

실습 목표

- 하나의 입력을 다양한 형식(문장형, 목록형, 표형식, JSON)으로 출력하도록 프롬프트를 조정하는 능력을 익힌다.
- Format 지정 여부에 따라 출력 구조가 어떻게 달라지는지 직접 비교한다.
- 출력 형식을 명확히 제어하는 것이 이후 에이전트·자동화 기능에서 왜 중요한지 이해한다.
- LLM의 “구조화된 출력 능력”을 활용하는 기본 기술을 익혀 후속 실습(함수 호출, 데이터 변환 등)으로 확장한다.

실습. 출력 형식(format)에 따른 결과 비교

```
from openai import OpenAI
from dotenv import load_dotenv
import os

load_dotenv()
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

topic = input("설명할 주제를 입력하세요: ")

formats = {
    "문장형": "2~3문장의 짧은 단락으로 설명하시오.",
    "목록형": "bullet 목록으로 4줄 이내로 설명하시오.",
    "표 형식": "선택한 내용을 2열 표 형식으로 정리하시오. (항목 | 설명)",
    "JSON": "다음을 JSON 형식으로 출력하시오. { '요약': '', '핵심포인트': [] }",
}

for fmt_name, fmt_instruction in formats.items():
    resp = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": "출력 형식을 정확히 지키는 AI입니다."},
            {"role": "user", "content": f"{topic}를 {fmt_instruction}"}
        ]
    )
```

```
    print(f"\n=== Format: {fmt_name} ===")
    print(resp.choices[0].message.content)
```

① 출력 형식에 따라 문장 구조와 정보 배열 방식이 완전히 달라진다

문장형은 자연스러운 서술로 구성되지만, 목록형은 핵심 포인트가 나열되며, JSON 형식은 명확한 키-값 구조를 갖춘다. 같은 내용을 요청하더라도 Format 지정 여부가 실제 출력의 품질을 결정한다.

② Format 기반 프롬프트는 품질 관리와 자동화에서 매우 중요하다

보고서 자동 생성, API 연동, 데이터 변환 등에서는 항상 일정한 형식이 필요하므로, Format 지시가 없으면 시스템 전체에 오류가 발생할 수 있다.

③ 모델의 "형식 준수 능력"을 적극적으로 활용해야 한다

LLM은 자연 언어뿐 아니라 구조적 출력에도 매우 능숙하므로, Format을 명확히 제시하면 원하는 수준의 일관성을 확보할 수 있다. 이후 자동화·에이전트·멀티포맷 생성 등 더 복잡한 기능의 기반이 된다.

3.2.3 Few-shot 및 예시 기반 프롬프트

① Few-shot Prompting은 "예시를 먼저 보여주고, 그 패턴을 따라 생성하도록 하는 방식"임

Few-shot 프롬프트는 모델에게 작업 예시를 1~3개 정도 제공하고, 그 예시의 형식과 흐름을 참고하여 새로운 결과를 생성하게 만드는 방법입니다. 모델은 예시를 단순한 텍스트가 아니라 **따라야 할 규칙의 샘플**로 해석하며, 그 패턴을 적용해 새로운 문장을 작성합니다. 이는 "이렇게 만들어라"라고 명확히 지시하는 효과가 있어, 초보자도 쉽게 결과 품질을 높일 수 있는 실용적인 프롬프트 기법입니다.

② 모델은 예시의 구조·톤·스타일을 그대로 모방하여 더 안정적인 결과를 생성함

예시 기반 프롬프트를 사용하면 모델이 작성해야 하는 문장의 형태, 문체, 길이, 표현 방식 등이 자연스럽게 통일됩니다. 예시가 없는 상태에서는 모델이 자유롭게 해석해 결과 편차가 생기지만, 예시를 제시하면 **출력의 일관성과 품질을 크게 높일 수 있습니다**.

예시는 다음과 같은 상황에서 특히 효과적입니다.

- 간결한 한 줄 문구 생성

- 보고서 같은 일정한 형식 유지
- 특정 문체 모방(친절한 톤, 간결한 톤, 마케팅 톤 등)
- “입력 → 출력” 규칙을 학습시키는 코드 생성

③ 콘텐츠 생성기·문장 변환기·기획서 자동화 등에서 매우 널리 사용되는 핵심 기법

SNS 콘텐츠 생성, 유튜브 스크립트 구성, 광고 문구 자동 생성 등에서는 **“패턴을 반복적으로 유지하는 능력”**이 매우 중요합니다. 이런 작업에서는 Few-shot이 특히 강력하게 작동해 자연스럽고 전문적인 결과를 만들게 됩니다. 또한 코드 생성에서도 입력과 출력 예시를 몇 개만 제공하면 모델이 규칙을 학습해 원하는 형태의 코드를 생성할 수 있어, 실무에서 가장 자주 사용되는 프롬프트 방식 중 하나입니다.

실습 목표

- 예시 제공 여부에 따라 모델이 생성하는 패턴이 어떻게 달라지는지 비교한다.
- Few-shot에서 “예시 패턴 반복”이 출력에 어떤 영향을 주는지 체험한다.
- 문장 패턴 모방, 문체 재현 등 구조적 생성 작업에서 Few-shot의 효과를 이해한다.

실습 1. 간단한 문장 패턴 모방 실험

아래 실습은 **예시 두 개의 문장 패턴을 따라 새로운 문장을 생성하는 실험**입니다.

현재 단계에서 가장 적합한 “입문용 Few-shot 실습” 형태입니다.

```
from openai import OpenAI
from dotenv import load_dotenv
import os

load_dotenv()
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

a = "AI 교육" # input("주제를 입력 하세요")
prompt = f"""
다음 예시 문장을 참고하여 같은 패턴의 새로운 문장을 작성하시오.

예시1: {a}은 미래 직업 역량을 키우는 데 필수적이다.
```

```
예시2: {a}은 문제 해결 능력을 강화하는 핵심 요소이다.

과제: '{a}의 사회적 가치'에 대해 같은 패턴으로 문장을 한 줄 작성하시오.
"""

resp = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": prompt}]
)

print("\n[출력 결과]")
print(resp.choices[0].message.content)
```

- 출력 문장이 **“AI 교육은 ~이다.”** 구조를 유지하는지 확인
- 예시 없이 요청했을 때와 비교하여 **구조·문체의 일관성 향상**을 직접 비교

실습 2. 문체(style) 모방 실험

이번 실습은 특정 문체를 하나 제시하고, 모델이 그 문체를 그대로 따라 글을 생성하는지 확인하는 실험입니다.

```
example_style = """
예시 문체:
'간결하고 직접적인 설명을 통해 독자가 핵심 내용을 빠르게 이해하도록 돕는다.'
"""

task = "생성형 AI의 장점을 같은 문체로 한 문단 작성하시오."

resp = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "user", "content": example_style + "\n" + task}
    ]
)

print("\n[출력 결과]")
print(resp.choices[0].message.content)
```

- 문장이 간결하고 직접적인 표현을 유지하는지 확인
- 문체·길이·톤이 예시 문체를 모방하고 있는지 비교

종합 확인 포인트

- **예시 제공 여부에 따라 결과의 안정성과 구조적 일관성이 크게 달라진다.**

  → 예시는 모델에게 "정답의 패턴"을 제공하는 강력한 힌트가 됨.

- **Few-shot은 실제 프로젝트에서 가장 자주 사용하는 프롬프트 패턴 중 하나이다.**

  → 기획서 자동 생성, 광고 문구 패턴 유지, 코드 생성 규칙 적용 등 모든 생성기에서 핵심 기능이 됨.

실습 3. 스탬프가 포함된 STT 실습

Whisper(Hugging Face)만 사용해 **타임스탬프가 포함된 STT CSV**를 만드는 실습.

(1) 필요한 패키지 설치

```
pip install --upgrade transformers accelerate torch soundfile pandas
```

※ `ffmpeg`는 설치되어 있지 안다면 아래 실습 4의 설치 방법을 참고 해서 설치합니다.

ffmpeg는 공식 배포 사이트에서 다운로드할 수 있습니다. 사용자는 아래 링크에서 최신 빌드를 내려받으면 됩니다.

https://www.gyan.dev/ffmpeg/builds/

해당 페이지에서 `ffmpeg-release-essentials.zip` 파일을 다운로드해서 `C:\ffmpeg\` 경로에 압축 해제.

(2) 단일 기능으로 다시 설계한 STT + 타임스탬프 코드

```
# stt_segments_whisper.py

import os
import torch
import pandas as pd
from dotenv import load_dotenv
```

```
from transformers import AutoModelForSpeechSeq2Seq, AutoProcessor, pi
peline

# ① 환경 설정
load_dotenv()
os.environ["PATH"] += os.pathsep + r"C:\ffmpeg\bin"  # ffmpeg 설치 경로

MODEL_ID = "openai/whisper-large-v3-turbo"


def create_asr_pipeline(model_id: str = MODEL_ID):
    """Whisper 모델과 Processor를 로드하고 STT 파이프라인을 생성합니다."""
    device = "cuda:0" if torch.cuda.is_available() else "cpu"
    torch_dtype = torch.float16 if torch.cuda.is_available() else torch.float32

    model = AutoModelForSpeechSeq2Seq.from_pretrained(
        model_id,
        torch_dtype=torch_dtype,
        low_cpu_mem_usage=True,
        use_safetensors=True,
    ).to(device)

    processor = AutoProcessor.from_pretrained(model_id)

    asr = pipeline(
        "automatic-speech-recognition",
        model=model,
        tokenizer=processor.tokenizer,
        feature_extractor=processor.feature_extractor,
        device=device,
        torch_dtype=torch_dtype,
        return_timestamps=True,  # 청크별 시작/끝 시간 포함
        chunk_length_s=10,
        stride_length_s=2,
    )

    return asr
```

```
def transcribe_with_timestamps(audio_path: str, csv_path: str) → pd.DataFr
ame:
    """오디오 파일을 받아 STT 결과를 (start, end, text) CSV로 저장합니다."""
    asr = create_asr_pipeline()

    print(f"[정보] STT 시작: {audio_path}")
    result = asr(audio_path)

    rows = []
    for chunk in result["chunks"]:
        start, end = chunk["timestamp"]
        text = chunk["text"].strip()
        rows.append((start, end, text))

    df = pd.DataFrame(rows, columns=["start", "end", "text"])
    df.to_csv(csv_path, index=False, sep="|", encoding="utf-8")

    print(f"[완료] CSV 저장: {csv_path}")
    return df


if __name__ == "__main__":
    # 같은 폴더에 있는 agent_output.mp3를 예시로 사용
    audio_file_path = "./agent_output.mp3"
    output_csv_path = "./agent_output_stt.csv"

    df = transcribe_with_timestamps(audio_file_path, output_csv_path)
    print(df.head())
```

이 코드의 특징은 다음과 같습니다.

① **pyannote, torchcodec, torchaudio 전부 제거**

② Hugging Face Whisper 파이프라인만 사용

③ `chunks`의 `timestamp`를 그대로 가져와 `start | end | text` CSV 생성

④ ffmpeg만 OS 차원에서 잘 잡혀 있으면, 나머지는 비교적 안정적으로 동작

실습 4. Whisper 기반 음성 인식 및 화자 분리 실습 과제

(실전 실력 향상을 위해 무작정 따라 해 보기)

① 실습 개요

이 실습에서는 Hugging Face의 Whisper 음성 인식 모델과 Pyannote 기반 화자 분리(diarization) 모델을 활용하여 하나의 오디오 파일에서 ① 시간대별 자막 텍스트를 추출하고, ② 화자별 발화 구간을 분석하는 과정을 경험합니다. Whisper를 통해 음성을 텍스트로 변환하고, Pyannote를 통해 누가 언제 말했는지를 구분함으로써, 실제 AI 서비스에서 사용되는 음성 처리 파이프라인의 기본 구조를 이해하는 것이 목표입니다.

② 실습 목표

1. Whisper STT(speech-to-text)의 입력·출력 흐름을 이해하고, 오디오 파일에서 시간 정보가 포함된 텍스트를 추출할 수 있습니다.
2. Pyannote 화자 분리 파이프라인을 사용하여 오디오 속 화자 구간을 RTTM 및 CSV 형식으로 분석할 수 있습니다.
3. Pandas를 활용해 화자별 구간을 그룹화하고, 시작·종료 시점과 발화 길이(duration)를 계산할 수 있습니다.
4. 전체 코드를 하나의 스크립트로 실행하여 실제 음성 처리 파이프라인을 완주하는 경험을 합니다.

③ 전체 소스 코드

먼저 아래의 필수 패키지를 가상 환경에 추가 설치 합니다.

① torch

Whisper 모델과 Pyannote 모델 로딩 시 사용됨. GPU 사용 시 CUDA 버전과 맞춰 설치 필요.

② transformers

Whisper STT 모델(Seq2Seq) 및 Processor 로딩에 필요.

③ pandas

STT 결과 및 화자 분리 결과를 CSV로 저장하는 데 사용.

④ pyannote.audio

화자 분리(speaker diarization) 기능 제공.

⑤ **soundfile**

Pyannote 또는 일부 오디오 전처리가 내부적으로 사용하는 라이브러리.

⑥ **librosa**

transformers Whisper 파이프라인 실행 시 내부적으로 필요할 수 있음(오디오 변환).

```
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu
pip install transformers==4.40.2
pip install accelerate
pip install datasets

pip install transformers
pip install pandas
pip install pyannote.audio
pip install soundfile
pip install librosa

pip install soundfile
```

추가 필수 요소(패키지는 아니지만 중요)

① ffmpeg 설치

Whisper는 오디오 파일을 처리할 때 ffmpeg가 시스템 PATH에 필요함.

이미 코드에 PATH 추가가 있으므로 설치만 되어 있으면 됨.

② Hugging Face 토큰

Pyannote diarization 모델은 Hugging Face 계정 토큰이 필요함.

---

ffmpeg.exe의 개념과 설치

① ffmpeg.exe의 개념

ffmpeg.exe는 오디오와 비디오 파일을 변환, 분석, 처리하기 위한 전용 프로그램입니다. 대부분의 음성·영상 관련 AI 라이브러리는 내부적으로 ffmpeg를 호출하여 파일을 디코딩하거나 형식을 변환합니다. 예를 들어 Whisper 음성 인식 모델은 mp3, m4a, wav 등 다양한 오디오 파일을 처리해야 하는데, Whisper 자체에는 파일 변환 기능이 없기 때문에

ffmpeg.exe를 통해 오디오 파일을 원하는 형식으로 변환하여 사용합니다. 따라서 ffmpeg는 음성 처리 파이프라인에서 필수적인 요소입니다.

② ffmpeg 설치가 필요한 이유

Python 패키지에는 ffmpeg가 포함되어 있지 않기 때문에 사용자가 별도로 설치해야 합니다. Whisper, Pyannote, transformers, librosa 등 여러 음성 처리 라이브러리는 내부적으로 다음과 같은 흐름으로 ffmpeg를 활용합니다.

> Whisper → ffmpeg.exe에 "오디오 파일을 16KHz wav로 변환하라"고 요청
> ffmpeg → 변환된 오디오 데이터를 Whisper에 전달
> Whisper → 최종적으로 음성 인식 결과(STT)를 생성

즉, ffmpeg가 설치되어 있지 않으면 Whisper는 오디오 파일을 정상적으로 읽을 수 없으며, 프로그램 실행 중 오류가 발생하게 됩니다.

③ ffmpeg 다운로드 방법

ffmpeg는 공식 배포 사이트에서 다운로드할 수 있습니다. 사용자는 아래 링크에서 최신 빌드를 내려받으면 됩니다.

https://www.gyan.dev/ffmpeg/builds/

해당 페이지에서 **ffmpeg-release-essentials.zip** 파일을 다운로드 해서 C:\ffmpeg\ 압축 해제 합니다.

Hugging Face 접근 권한(Accept)

① Hugging Face 로그인

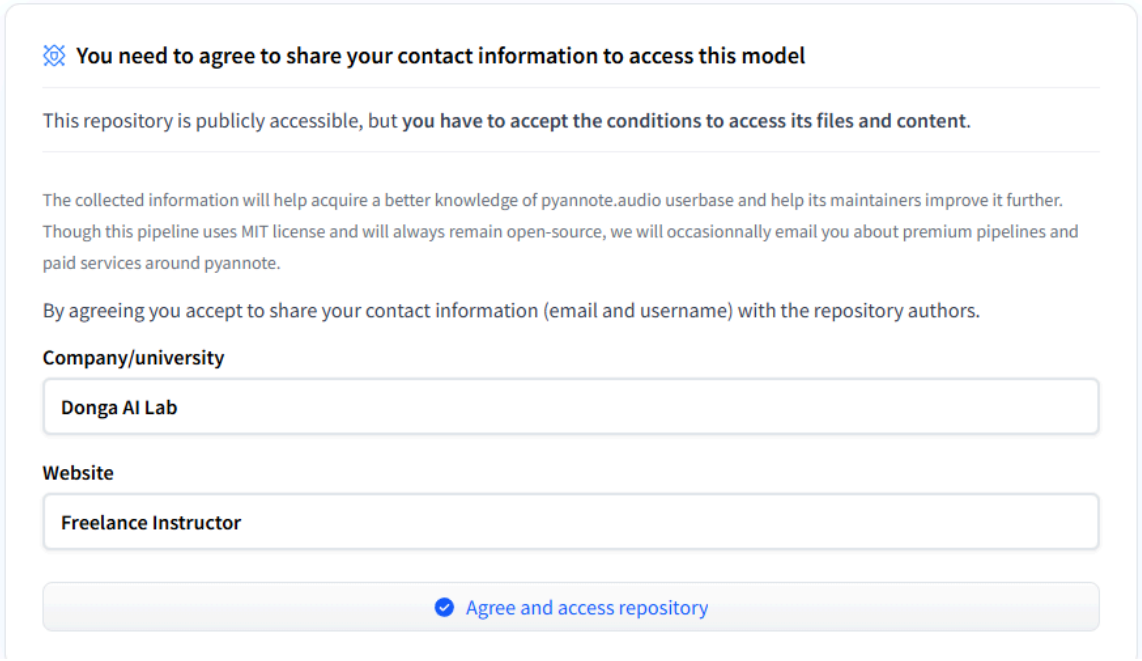
② 해당 모델 접근 권한(Accept)

③ 올바른 토큰을 코드에 연결

Step 1: Hugging Face에서 모델 조건 수락

1. 브라우저에서 Hugging Face에 로그인합니다.
2. 아래 주소로 이동합니다. (3.1, 3.0 등 Gate된 세 곳 모두 Access 동의 해야 함)
    - `https://huggingface.co/pyannote/speaker-diarization-3.1`
    - `https://huggingface.co/pyannote/segmentation-3.0`
    - `https://huggingface.co/pyannote/speaker-diarization-community-1`

3. 페이지에 들어가면, 중간쯤에
    - `Agree and access repository` 같은 버튼 또는 체크박스가 있을 것입니다.
    - 여기에 예를 들어 아래처럼 입력.
    - `Donga AI Lab` 또는실제 소속 학교/기관명 프리랜서라면 `Individual / Freelance Instructor` 등 입력.
    - website에는 그냥 블로그나 깃허브 주소 넣으면 됨. (아무거나 넣어도 됨)



4. 해당 조건에 동의(accept)해야 이 계정으로 이 모델을 쓸 수 있습니다.

이 과정을 안 하면, **아무 토큰을 써도 401이 계속 뜹니다.**

Step 2: 토큰 발급 또는 확인

1. Hugging Face 우측 상단 프로필 → `Settings` → `Access Tokens` 메뉴로 이동합니다.
2. 이미 토큰을 하나 만들어 두셨다면, 그 토큰의 **Scope(권한)**에 최소한 `read` 권한이 있는지 확인합니다.
3. 없다면 **새 토큰(New token)**을 생성합니다.
    - 이름: `pyannote-token` 정도
    - 권한: 최소 `Read` (읽기) 체크

생성된 토큰 문자열: `hf_XXXXXXXXXXXX...` 형태입니다.

Hugging Face 토큰 준비

Hugging Face 로그인 →

👉 https://huggingface.co/settings/tokens

여기에서 `Read` **권한 토큰**을 생성합니다.

Step 3: 토큰을 코드/환경에 제대로 연결하기

`.env` 파일에 다음처럼 추가합니다.

```
HF_TOKEN=hf_여기에_hf_토큰값
```

소스 코드

아래 코드를 `stt_diarization.py`와 같은 파일로 저장한 뒤, 오디오 파일 경로만 자신의 환경에 맞게 수정하여 실행합니다. (의존성 문제로 오류가 많이 나는 소스입니다.)

```
import os
import torch
import pandas as pd
from transformers import AutoModelForSpeechSeq2Seq, AutoProcessor, pipeline
from pyannote.audio import Pipeline

from dotenv import load_dotenv
import soundfile as sf   # ← torchaudio 대신 soundfile 사용

load_dotenv()

os.environ["PATH"] += os.pathsep + r"C:\ffmpeg\bin"  # 자신이 설치한 ffmpeg 경로로 수정

def whisper_stt(
    audio_file_path: str,
    output_file_path: str = "./output.csv"
):
    device = "cuda:0" if torch.cuda.is_available() else "cpu"
    torch_dtype = torch.float16 if torch.cuda.is_available() else torch.float32
```

```
    model_id = "openai/whisper-large-v3-turbo"

    model = AutoModelForSpeechSeq2Seq.from_pretrained(
        model_id,
        torch_dtype=torch_dtype,
        low_cpu_mem_usage=True,
        use_safetensors=True
    )
    model.to(device)

    processor = AutoProcessor.from_pretrained(model_id)

    pipe = pipeline(
        "automatic-speech-recognition",
        model=model,
        tokenizer=processor.tokenizer,
        feature_extractor=processor.feature_extractor,
        torch_dtype=torch_dtype,
        device=device,
        return_timestamps=True,  # 청크별로 타임스탬프를 반환
        chunk_length_s=10,       # 입력 오디오를 10초씩 나누기
        stride_length_s=2,       # 2초씩 겹치도록 청크 나누기
    )

    result = pipe(audio_file_path)
    df = whisper_to_dataframe(result, output_file_path)

    return result, df


def whisper_to_dataframe(result, output_file_path):
    start_end_text = []

    for chunk in result["chunks"]:
        start = chunk["timestamp"][0]
        end = chunk["timestamp"][1]
        text = chunk["text"].strip()
        start_end_text.append([start, end, text])
```

```
    df = pd.DataFrame(start_end_text, columns=["start", "end", "text"])
    df.to_csv(output_file_path, index=False, sep="|")
    return df


def speaker_diarization(
    audio_file_path: str,
    output_rttm_file_path: str,
    output_csv_file_path: str
):
    # ① Hugging Face 토큰
    hf_token = os.getenv("HF_TOKEN")
    print(">>>>>>>>>>>>> HF_TOKEN in code:", hf_token)

    pipeline = Pipeline.from_pretrained(
        "pyannote/speaker-diarization-3.1",
        token=hf_token
    )

    # ② cuda 사용 여부
    if torch.cuda.is_available():
        pipeline.to(torch.device("cuda"))
        print("cuda is available")
    else:
        print("cuda is not available")

    # ③ soundfile로 오디오 로드
    audio, sample_rate = sf.read(audio_file_path)

    # (time,) 또는 (time, channels) → (channels, time)
    if audio.ndim == 1:          # 모노
        audio = audio[None, :]   # (1, time)
    else:                        # 스테레오 이상
        audio = audio.T          # (channels, time)

    waveform = torch.from_numpy(audio).float()
```

```
file_dict = {
    "waveform": waveform,
    "sample_rate": sample_rate,
}

# ④ 화자 분리 실행
out = pipeline(file_dict)                  # DiarizeOutput 객체
ann = out.speaker_diarization              # Annotation 객체

# ⑤ RTTM 형식으로 저장 (이제 ann.write_rttm 사용)
with open(output_rttm_file_path, "w", encoding="utf-8") as rttm:
    ann.write_rttm(rttm)

# ⑥ RTTM → DataFrame 변환 (이 아래는 그대로 사용)
df_rttm = pd.read_csv(
    output_rttm_file_path,
    sep=" ",
    header=None,
    names=[
        "type", "file", "chnl", "start", "duration",
        "C1", "C2", "speaker_id", "C3", "C4",
    ],
)

df_rttm["end"] = df_rttm["start"] + df_rttm["duration"]

df_rttm["number"] = None
df_rttm.at[0, "number"] = 0

for i in range(1, len(df_rttm)):
    if df_rttm.at[i, "speaker_id"] != df_rttm.at[i - 1, "speaker_id"]:
        df_rttm.at[i, "number"] = df_rttm.at[i - 1, "number"] + 1
    else:
        df_rttm.at[i, "number"] = df_rttm.at[i - 1, "number"]

df_rttm_grouped = df_rttm.groupby("number").agg(
    start=pd.NamedAgg(column="start", aggfunc="min"),
    end=pd.NamedAgg(column="end", aggfunc="max"),
```

```
        speaker_id=pd.NamedAgg(column="speaker_id", aggfunc="first"),
    )

    df_rttm_grouped["duration"] = (
        df_rttm_grouped["end"] - df_rttm_grouped["start"]
    )

    df_rttm_grouped.to_csv(
        output_csv_file_path,
        index=False,
        encoding="utf-8",
    )

    return df_rttm_grouped


if __name__ == "__main__":
    # 같은 폴더에 있는 agent_output.mp3 사용
    audio_file_path = "./agent_output.mp3"            # 원본 오디오 파일

    # 출력 파일들도 굳이 chap05 폴더 말고, 현재 폴더에 두셔도 됩니다.
    stt_output_file_path = "./agent_output_stt.csv"    # STT 결과 파일
    rttm_file_path = "./agent_output.rttm"            # 화자 분리 RTTM 파일
    rttm_csv_file_path = "./agent_output_rttm.csv"     # 화자 분리 CSV 파일

    # ① Whisper STT 실행 (주석 해제 후 사용)
    # result, df = whisper_stt(
    #     audio_file_path,
    #     stt_output_file_path
    # )
    # print(df)

    # ② 화자 분리 실행
    df_rttm = speaker_diarization(
        audio_file_path,
        rttm_file_path,
        rttm_csv_file_path
    )
```

```
print(df_rttm)
```

④ 단계별 실습 절차

1. ffmpeg 설치 경로가 올바르게 설정되어 있는지 확인하고, `os.environ["PATH"] += ...` 부분을 자신의 시스템 경로에 맞게 수정합니다.
2. Hugging Face 토큰을 발급받아 `speaker_diarization` 함수의 `use_auth_token` 인자에 올바르게 입력합니다.
3. `audio_file_path`를 실제 보유한 음성 파일 경로로 변경합니다. 예시와 같이 `chap05/audio` 폴더를 사용해도 되고, 자신의 테스트용 mp3 파일을 사용해도 됩니다.
4. 처음에는 Whisper STT 부분을 주석 처리한 상태로 화자 분리만 실행해 보고, 이후 주석을 해제하여 STT 결과 CSV도 함께 생성해 봅니다.
5. 생성된 CSV 파일 두 개(STT 결과, 화자 분리 결과)를 열어 시작 시각, 종료 시각, 화자 ID, 텍스트가 어떤 식으로 정리되어 있는지 직접 확인합니다.

⑤ 확인 포인트

1. Whisper STT 결과 CSV에서 `start`, `end`, `text` 컬럼이 시간 순서대로 올바르게 채워져 있는지 확인합니다.
2. 화자 분리 결과 CSV에서 `speaker_id`별로 `start`, `end`, `duration`이 계산되어 있는지 확인합니다.
3. 동일한 오디오 파일에 대해 STT와 화자 분리 결과를 비교하면, 어느 구간에서 어떤 화자가 어떤 내용을 말했는지 추론할 수 있음을 이해합니다.
4. 이 파이프라인이 실제 회의록 자동 생성, 상담 콜 분석, 팟캐스트 편집 자동화 등의 서비스로 어떻게 확장될 수 있을지 생각해 봅니다.

⑥ 도전 과제

1. STT CSV와 화자 분리 CSV를 Pandas에서 병합하여 “화자별 자막 스크립트” 형태의 결과를 만들어 봅니다.
2. 화자별 `duration` 합계를 계산하여, 전체 발화 시간에서 각 화자가 차지하는 비율을 구해 보고 그래프로 시각화해 봅니다.
3. `chunk_length_s`와 `stride_length_s` 값을 변경하면서 인식 품질과 처리 시간의 변화를 비교해 보고, 자신이 사용하는 오디오 유형에 적절한 값을 탐색해 봅니다.