생성형 AI 기반 RAG 개발자 과정

OpenAI API 첫걸음

Session 4 / 33 — Day 1

REST · SDK old · SDK new · stateless · tokens

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

이번 시간을 마치면

개념

- ✅ LLM API 가 **HTTP req/resp** 일 뿐임을 이해한다
- ✅ API 는 **stateless** — 누가 누군지 모르고, 대화 연속성 없다
- ✅ **토큰** 이 곧 비용 단위임을 알고 호출 전에 추정할 수 있다

실습

- ✅ **OpenAI** 가입 + **API Key 발급** + Hard limit 설정
- ✅ **REST API 직접 호출** (`requests.post`) — `1.intro/1.restapi.py`
- ✅ **SDK old / new** 비교 실행 — `2.sdk_old.py` , `3.sdk_new.py`
- ✅ 주요 **파라미터 4개** 튜닝 — temperature · top_p · penalties · max_tokens
- ✅ **tiktoken** 으로 호출 전 비용 추정

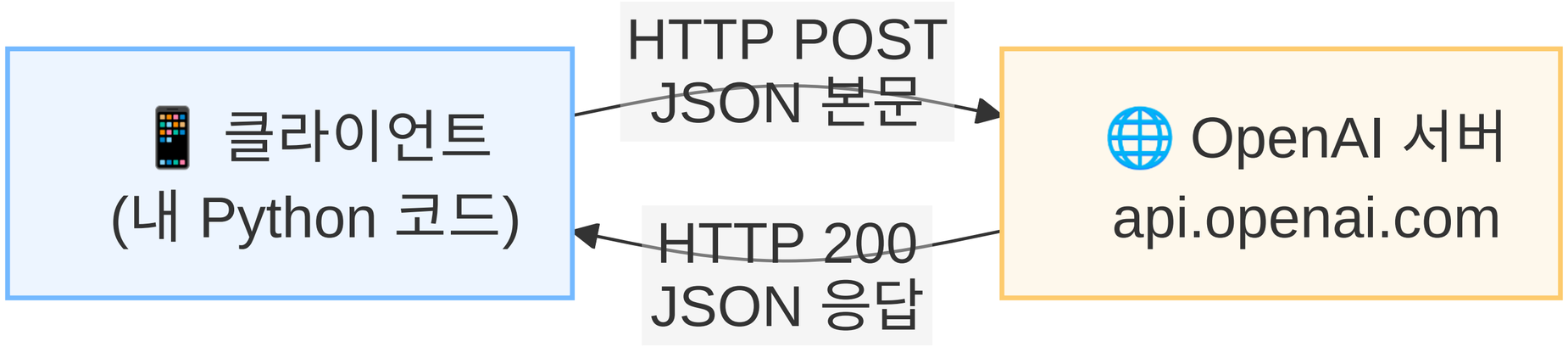
> 📂 베이스 소스: `tutorial-genai/1.openai/1.intro/`

PART A

PART A
API 의 본질

> stateless · 연속성 없음 · 토큰 단위 — 약 15분

API = "프로그램이 부르는 웹 페이지"



웹 브라우저	LLM API
URL 입력 → GET 요청	`https://api.openai.com/v1/chat/completions` 로 POST
HTML 응답 → 화면 렌더	JSON 응답 → 본문에서 `choices[0].message.content` 추출
사용자: 사람	사용자: **내 Python 코드**

> 💡 OpenAI/Anthropic API 는 **REST 위에 올린 LLM 서비스**입니다. 본질은 HTTP.

API 는 stateless — 매번 처음 보는 사람

❌ 일반인의 오해

> "ChatGPT 웹사이트처럼 대화를 기억할 것이다"

```
# 1번째 호출
client.chat.completions.create(
    messages=[{"role":"user","content":"내 이름은 홍길동"}]
)

# 2번째 호출 (1번째와 다른 호출)
client.chat.completions.create(
    messages=[{"role":"user","content":"내 이름이 뭐였지?"}]
)
# → "죄송하지만 알지 못합니다" 😞
```

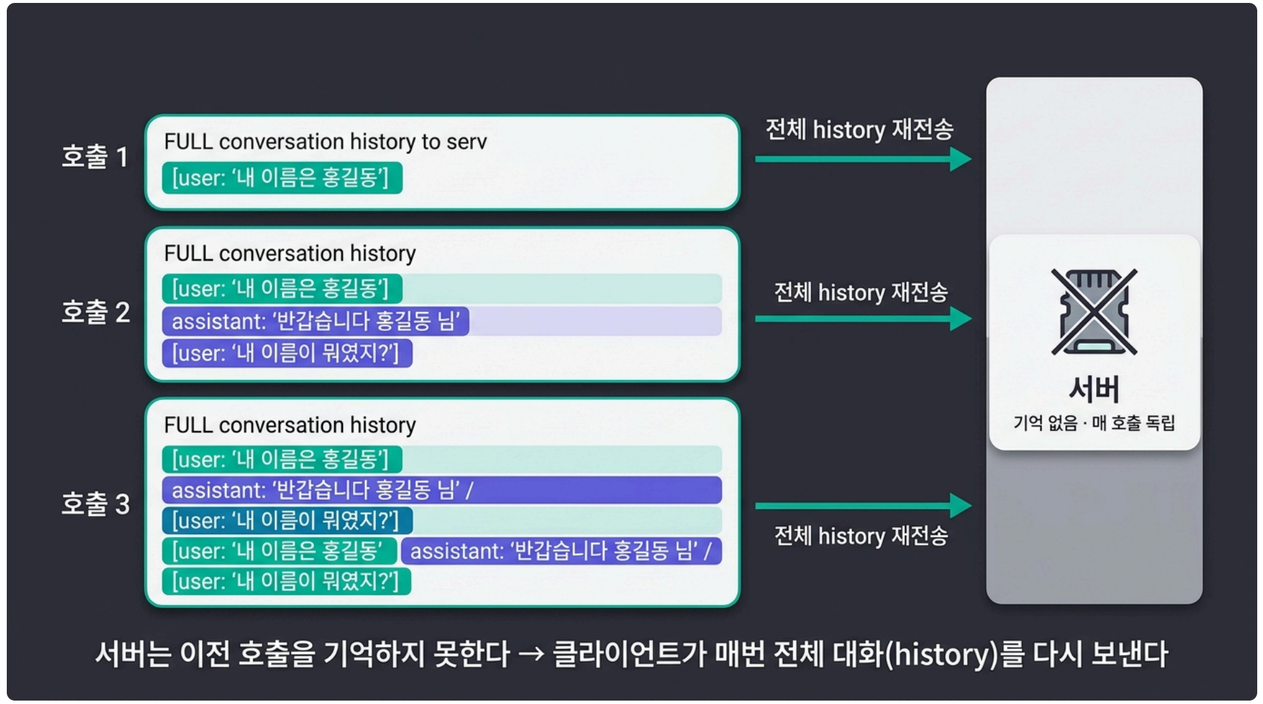
✅ 실제 동작

- 모든 호출은 **독립**
- 서버는 이전 호출을 기억하지 않음
- "대화" 처럼 보이려면 **클라이언트가 직접 messages 배열에 과거를 누적**

```
history = [
    {"role":"user","content":"내 이름은 홍길동"},
    {"role":"assistant","content":"반갑습니다 홍길동 님"},
    {"role":"user","content":"내 이름이 뭐였지?"},
]
client.chat.completions.create(messages=history)
# → "홍길동 님이라고 하셨습니다" ✅
```

> 🎯 **Session 5 의 핵심**: 이 "history 관리" 가 챗봇 코드의 절반을 차지합니다.

매 호출마다 전체 대화를 다시 보낸다



💡 서버는 이전 호출을 **기억하지 못합니다.** "대화"처럼 보이려면 클라이언트가 매 호출마다 **과거 전체(history)를 다시 전송**해야 합니다.

토큰 (Token) — 글자도 단어도 아닌 단위

토큰화 예시

입력	토큰
`Hello`	1 토큰
`Hello world`	2 토큰
`안녕하세요`	약 5~7 토큰 (한글은 비효율)
`tokenization`	2 토큰 (`token` + `ization`)

한 줄 규칙

- **영어**: 1 단어 ≈ 1.3 토큰
- **한글**: 1 글자 ≈ 1.5~2 토큰
- **공백·특수문자**: 별도 토큰

tiktoken 으로 직접 계산

```
# pip install tiktoken
import tiktoken

enc = tiktoken.encoding_for_model("gpt-4o-mini")

def count(text: str) → int:
    return len(enc.encode(text))

print(count("Hello world"))          # 2
print(count("안녕하세요"))           # 6
print(count("토큰화 예시입니다"))    # 10
```

비용 = (입력 토큰 + 출력 토큰) × 모델 단가

- 입력 1M 토큰 ≈ 0.15달러 (gpt-4o-mini)
- 출력 1M 토큰 ≈ 0.60달러
- 출력이 보통 4~5배 비쌈

> 💡 **호출 전에 토큰을 셀 수 있다** = **호출 전에 비용을 알 수 있다**.

PART B

PART B
OpenAI 가입과 키 발급

실제 화면 흐름 — 약 10분

1단계 — openai.com 접속

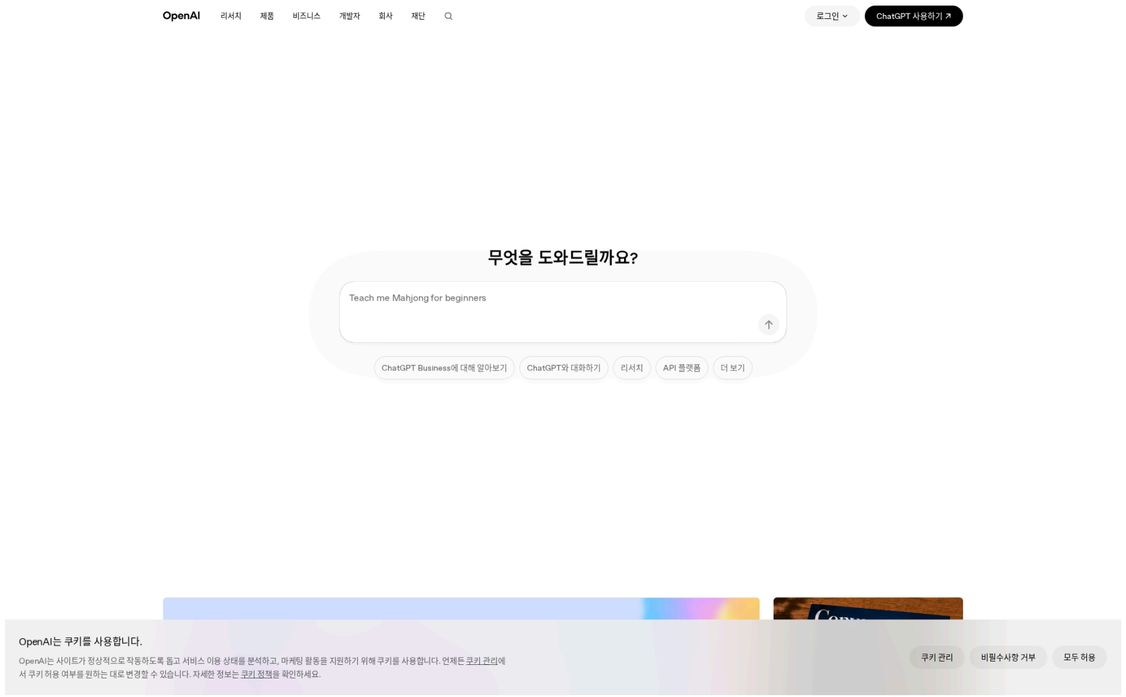
경로

- 메인 → 우측 상단 **"Log in"** 또는 **"Sign up"**
- **개발자** 라면 우상단 **"API"** 또는 **platform.openai.com** 직접 접속

ChatGPT vs API 플랫폼

사이트	용도
`chat.openai.com`	일반 ChatGPT 사용 (구독제)
`platform.openai.com`	개발자 API · 결제 · 키 관리

> 📌 같은 인증을 쓰지만 **결제·키는 플랫폼 쪽에서 따로** 관리합니다.

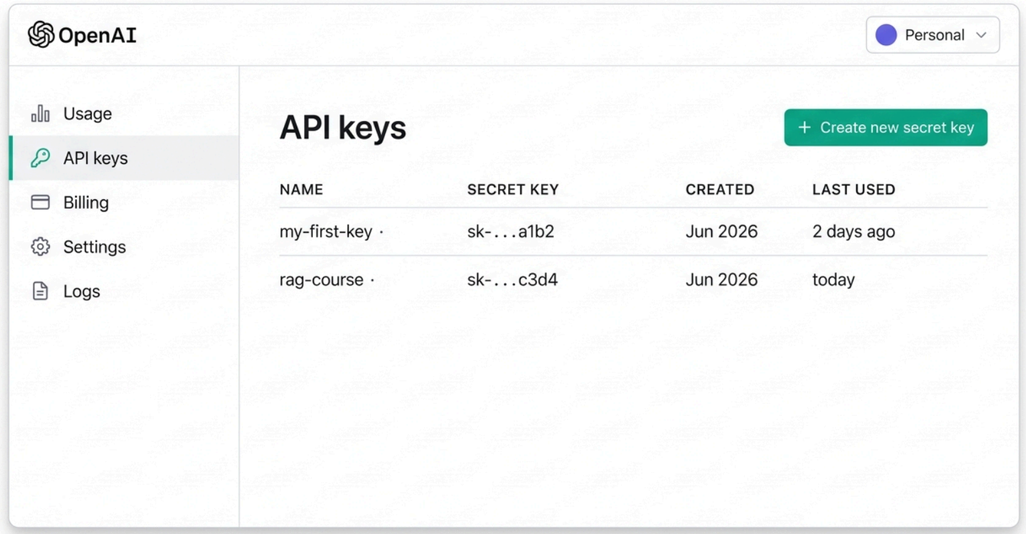


2단계 — platform.openai.com (개발자 콘솔)

사이드바 주요 메뉴

메뉴	역할
API keys	키 발급·폐기·이름 부여
Usage	호출 횟수·토큰·달러 사용량
Billing	결제수단·한도 (Soft/Hard limit)
Settings	조직·사용자·모델 접근권한
Logs	최근 호출 내역과 비용

> 💡 회사 계정과 개인 계정은 **Organization** 으로 분리. 코스에서는 개인 계정 권장.



3단계 — API Key 발급

절차

1. `platform.openai.com/api-keys` 접속
2. 우측 상단 **"Create new secret key"** 클릭
3. **이름** 부여 (예: `genai-course-2026`) — 추후 식별용
4. **Permissions** 선택 (기본값 "All" 로 OK)
5. **Create** → 키가 한 번만 표시됨 → 즉시 복사

⚠️ 보안 주의

- **이 화면을 닫으면 키를 다시 볼 수 없습니다** (재발급만 가능)
- 키를 즉시 `.env` 에 저장
- 절대 스크린샷·슬랙·코드 본문에 노출 금지

> 🖼️ **강사용 안내**: API Keys 페이지는 로그인 후에만 보입니다. 강사 계정으로 위 단계를 실시간 시연 또는 캡처 → `slides_assets/captures/openai_api_keys_placeholder.png` 로 덮어 쓰면 이 슬라이드에 자동 반영.

4단계 — 가격 정책 확인

본 코스에서 권장하는 모델 (2026 기준)

모델	입력 1M	출력 1M	용도
gpt-4o-mini	$0.15	$0.60	강의 시연 · 챗봇 · 분류 — 가성비 최고
gpt-4o	$2.50	$10.00	복잡한 추론·정확도 필요 시
text-embedding-3-small	$0.02	—	Day 2 임베딩
text-embedding-3-large	$0.13	—	정밀 RAG 가 필요할 때

> 📌 **요금은 수시 변동** — 강의일에 openai.com/api/pricing 에서 재확인.

5단계 — 어떤 모델이 있나

모델군 (2026)

패밀리	대표 모델	강점
GPT-4o	gpt-4o, gpt-4o-mini	범용 chat · vision · tool use
GPT-4.1	gpt-4.1, gpt-4.1-mini	코딩·instruction following
o1 / o3	o1, o3-mini	깊은 추론 (느리지만 정확)
Embeddings	text-embedding-3-small/large	RAG · 유사도 검색
Whisper	whisper-1	음성 → 텍스트
DALL-E / GPT-Image	dall-e-3, gpt-image-1	이미지 생성

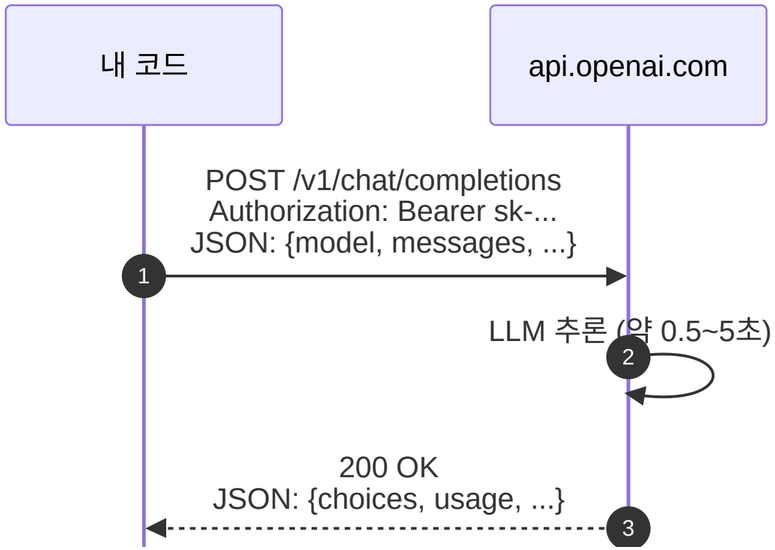
> 🎯 **본 코스의 기본은** `gpt-4o-mini`. 정밀도 필요할 때만 더 큰 모델로 업그레이드.

PART C

PART C
REST API 직접 호출

> SDK 없이 — requests 만으로 — 약 15분

한 번의 호출 = 한 번의 HTTP POST



호출 구성 4가지

구성	값
메서드	**POST**
URL	`https://api.openai.com/v1/chat/completions`
인증	`Authorization: Bearer sk-...`
본문	`{"model": "gpt-4o-mini", "messages": [...], "temperature": 0.7}`

SDK 없이 — curl 만

```
curl https://api.openai.com/v1/chat/completions \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gpt-4o-mini",
    "messages": [{"role": "user", "content": "한 줄 인사"}],
    "max_tokens": 30
  }'
```

응답 (요약)

```
{
  "id": "chatcmpl-...",
  "model": "gpt-4o-mini-2024-07-18",
  "choices": [{
    "index": 0,
    "message": {"role": "assistant", "content": "안녕하세요! 무엇을 도와드릴까요?"},
    "finish_reason": "stop"
  }],
  "usage": {"prompt_tokens": 10, "completion_tokens": 12, "total_tokens": 22}
}
```

> 🎯 **포인트**: SDK 가 하는 일은 결국 위 HTTP 요청을 만드는 것뿐.

1.restapi.py — 가장 본질에 가까운 호출

```
# tutorial-genai/1.openai/1.intro/1.restapi.py
import requests, os
from dotenv import load_dotenv

load_dotenv(dotenv_path='../.env')
api_key = os.getenv('OPENAI_API_KEY')

def ask_chatgpt(user_input: str) → str:
    resp = requests.post(
        'https://api.openai.com/v1/chat/completions',
        json={
            'model': 'gpt-4o-mini',
            'messages': [
                {'role': 'system', 'content': 'You are a helpful assistant.'},
                {'role': 'user',   'content': user_input},
            ],
            'max_tokens': 100,
            'temperature': 0.7,
        },
        headers={
            'Content-Type':  'application/json',
            'Authorization': f'Bearer {api_key}',
        },
    )
```

💡 회사망/프록시 환경에서 SSL 이슈 나면 `requests.post(..., verify=False)` 임시 우회 (운영 환경에선 금지).

응답에서 챙길 4가지

```
{
  "id": "chatcmpl-9abcDEF...",          // 호출 식별자 (디버그·과금 추적)
  "model": "gpt-4o-mini-2024-07-18",    // 실제 사용 모델 (-mini 도 버전 명시)
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "dict 는 키·값 쌍, set 은 값만"
      },
      "finish_reason": "stop"            // stop / length / content_filter / tool_calls
    }
  ],
  "usage": {
    "prompt_tokens": 28,                 // 보낸 토큰
    "completion_tokens": 14,             // 받은 토큰
    "total_tokens": 42                   // 합 (이것이 곧 청구 단위)
  }
}
```

코드에서 항상 확인할 3가지

1. `choices[0].finish_reason == "stop"` — 정상 종료 (length 면 max_tokens 부족)
2. `usage.total_tokens` — 비용 추적
3. `model` — 실제 라우팅된 모델 (별칭 → 구체 버전)

PART D

PART D
SDK — old vs new

> 두 버전이 공존하는 이유 — 약 15분

2.sdk_old.py — 구버전 (openai==0.28)

```
# pip install openai==0.28
import openai, os
from dotenv import load_dotenv

load_dotenv(dotenv_path='../.env')
openai.api_key = os.getenv('OPENAI_API_KEY')   # ← 모듈 전역에 키

def ask_chatgpt(user_input: str) → str:
    response = openai.ChatCompletion.create(   # ← 클래스메서드 스타일
        model='gpt-3.5-turbo',
        messages=[
            {'role': 'system', 'content': 'You are a helpful assistant.'},
            {'role': 'user',   'content': user_input},
        ],
    )
```

특징

- **모듈 전역 상태** — `openai.api_key` 에 키를 설정
- `openai.ChatCompletion.create(...)` — 클래스 단위 호출
- 응답은 **dict** — `response['choices'][0]['message']['content']`

> ⚠️ **2024-11 이후 deprecated**. 구 코드를 만나면 익숙해질 필요는 있지만 새 코드는 SDK new 로.

3.sdk_new.py — 신버전 (openai>=1.0)

```
# pip install openai==1.*
import openai, os
from dotenv import load_dotenv

load_dotenv(dotenv_path='../.env')

client = openai.OpenAI(api_key=os.getenv('OPENAI_API_KEY'))   # ← 클라이언트 인스턴스

def ask_chatgpt(user_input: str) → str:
    response = client.chat.completions.create(                # ← 인스턴스 메서드
        model='gpt-4o-mini',
        messages=[
            {'role': 'system', 'content': 'You are a helpful assistant.'},
```

특징

- **클라이언트 인스턴스** — 키는 인스턴스 단위, 멀티 키·멀티 환경 깔끔
- `client.chat.completions.create(...)` — 인스턴스 메서드
- 응답은 **Pydantic 객체** — `.choices[0].message.content` (IDE 자동완성)

> 🔐 **키는 코드에 직접 적지 말 것** — `.env` 에 두고 `.gitignore` 등록, `os.getenv("OPENAI_API_KEY")` 로만 읽기. 모델명도 `MODEL = "gpt-4o-mini"` 같은 **상수/환경변수**로 관리해 한 곳에서 교체.

세 방식 비교

항목	REST (requests)	SDK old	SDK new
의존성	`requests`	`openai==0.28`	`openai>=1.0`
호출 방식	`requests.post(url, json=...)`	`openai.ChatCompletion.create(...)`	`client.chat.completions.create(...)`
키 관리	헤더에 직접	`openai.api_key = ...` (전역)	`OpenAI(api_key=...)` (인스턴스)
응답 접근	`r.json()['choices'][0]['message']['content']`	`r['choices'][0]['message']['content']`	`r.choices[0].message.content`
타입 힌트	dict	dict	**Pydantic 객체** (자동완성 ✅)
비동기	직접 처리	미흡	`AsyncOpenAI` 기본 지원
스트리밍	가능 (직접 파싱)	가능	**권장** (chunks iterator)
권장도	학습·디버그용	레거시 유지보수	**현행 신규 코드**

마이그레이션 가이드 (구→신)

```
# Before (old)
openai.api_key = key
r = openai.ChatCompletion.create(model=..., messages=...)
text = r['choices'][0]['message']['content']

# After (new)
client = openai.OpenAI(api_key=key)
r = client.chat.completions.create(model=..., messages=...)
```

PART E

PART E
주요 파라미터 튜닝

temperature · top_p · penalties · max_tokens — 약 10분

3가지 role — 대화의 등장인물

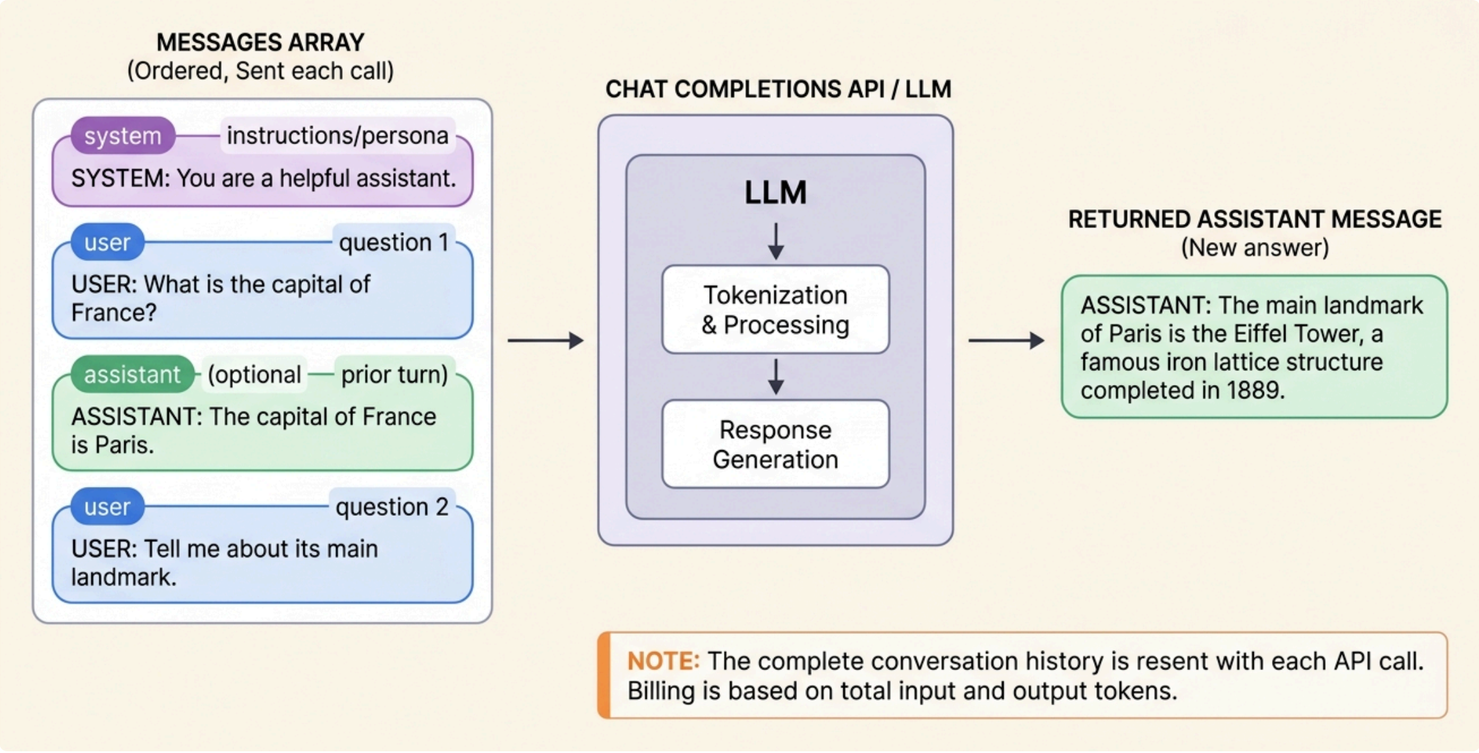
role	누가 작성	용도
system	개발자	"당신은 …이다" — 페르소나·규칙 (대화 시작 1번)
user	사용자 (또는 코드가 사용자 역할로)	질문·지시
assistant	모델 응답	과거 응답을 messages 에 다시 넣어 "기억" 시뮬레이션

일반 패턴

```
messages = [
    {"role": "system",    "content": "한국어로만 답하는 Python 전문가."},
    {"role": "user",      "content": "리스트 컴프리헨션이 뭐예요?"},
    {"role": "assistant", "content": "리스트를 한 줄로 만드는 …"},   # ← 과거 응답 (옵션)
    {"role": "user",      "content": "예시 코드 하나"},
]
```

> 💡 **회차 5 의 핵심**: 챗봇은 위 messages 배열을 매 호출마다 누적 관리하는 코드. 누적된 **이력 전체가 매 호출 입력 토큰으로 다시 과금**됩니다 — 대화가 길수록 비용 증가.

Chat Completion — messages 배열을 보내면 다음 메시지가 나온다



Chat Completion · system·user·assistant messages 배열을 보내면 다음 assistant 메시지 생성(매 호출 이력 전체 전송·토큰 과금)

temperature (0.0 ~ 2.0) — 무작위성

값	효과	적합한 작업
0.0	거의 결정적 — 같은 입력에 같은 출력	코드 생성·분류·구조화 출력
0.2~0.4	약간의 변동 — 자연스럽되 일관성 유지	요약·번역·FAQ 답변
0.7 (기본)	균형 — 적당히 창의적	일반 챗봇
1.0~1.5	창의적 — 같은 입력에 다른 답	아이디어·이야기·카피라이팅
2.0	거의 랜덤 — 일관성 거의 없음	실험적 출력 (실무에서는 거의 안 씀)

실험

```
for t in [0.0, 0.7, 1.5]:
    r = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[{'role':'user','content':'커피 카페 이름 1개'}],
        temperature=t, max_tokens=10,
    )
    print(f"t={t} → {r.choices[0].message.content}")
```

> 🎯 **분류·추출**: 0.0 · **자연어 답변**: 0.7 · **창작**: 1.0+

top_p (0.0 ~ 1.0) — Nucleus Sampling

의미

- 다음 토큰 후보 중 **누적 확률 p 까지의 토큰만** 후보로 둠
- `top_p=0.9` → 누적 90% 까지의 토큰 중에서 샘플링
- `top_p=0.1` → 가장 확률 높은 몇 개만

temperature 와 관계

- 둘 다 **무작위성** 조절
- 한 번에 **둘 중 하나만** 조절 권장 (혼동 방지)
- 일반적으로 `temperature` 만 만지면 충분

언제 top_p 를 쓰나

- **품질 최상위만 원할 때** → `top_p=0.1, temperature=1.0`
- **다양성·창의성** → `top_p=0.95, temperature=0.9`

권장 기본값

```
# 일반
temperature=0.7

# 더 안정적
temperature=0.2, top_p=1.0

# 창의적
temperature=1.0, top_p=0.95
```

응답 길이와 다양성 제어

파라미터	범위	효과
max_tokens	1 ~ 모델 한도	응답 최대 토큰. 비용 상한선 — 반드시 설정
frequency_penalty	-2.0 ~ 2.0	양수: 같은 단어 반복 억제 / 음수: 반복 유도
presence_penalty	-2.0 ~ 2.0	양수: 새로운 주제 도입 장려 / 음수: 주제 고정
stop	string 또는 리스트	해당 문자열 나오면 응답 즉시 종료 (예: `"\n\n"`)

권장 시작점 (실무 기본)

```
client.chat.completions.create(
    model='gpt-4o-mini',
    messages=[...],
    max_tokens=500,          # 답변 길이 상한
    temperature=0.7,
    frequency_penalty=0.0,   # 처음엔 손대지 말 것
    presence_penalty=0.0,
)
```

> ⚠️ **max_tokens 미설정 시** 모델이 한도까지 가버려서 비용·시간 폭증. 항상 설정.

PART F

PART F
토큰과 비용

> 호출 전에 가격을 알자 — 약 10분

호출 전 비용 추정

```python
import tiktoken

def estimate_cost(model: str, prompt: str, expected_output_tokens: int = 200) → float:
    """예상 비용을 달러로 반환 — 2026 기준."""
    PRICING = {  # 입력 / 출력 단가 (1M 토큰)
        "gpt-4o-mini": (0.15, 0.60),
        "gpt-4o":      (2.50, 10.00),
    }
    enc = tiktoken.encoding_for_model(model)
    prompt_tokens = len(enc.encode(prompt))
    p_in, p_out = PRICING[model]
    cost = (prompt_tokens * p_in + expected_output_tokens * p_out) / 1_000_000
    return cost

# 사용 예
prompt = "Python list comprehension 을 5줄 설명하세요" * 10
print(f"입력 토큰: {len(tiktoken.encoding_for_model('gpt-4o-mini').encode(prompt))}")
print(f"예상 비용: ${estimate_cost('gpt-4o-mini', prompt, 300):.6f}")
```

실행 결과

```
입력 토큰: 130
예상 비용: $0.000200
```

> 💡 1만 회 호출해도 $2. 카페 한 잔. 단, 모델·길이가 커지면 빠르게 증가.

호출 후 실제 토큰을 기록

```
total_in, total_out = 0, 0

def ask(msg: str) → str:
    global total_in, total_out
    r = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[{'role':'user','content':msg}],
        max_tokens=200,
    )
    total_in  += r.usage.prompt_tokens
    total_out += r.usage.completion_tokens
    return r.choices[0].message.content

# 100회 호출 후 누적 비용 계산
for i in range(100):
    ask(f"숫자 {i} 을 영문으로")

cost = (total_in * 0.15 + total_out * 0.60) / 1_000_000
print(f"누적: in={total_in:,}, out={total_out:,}, cost=${cost:.4f}")
```

🎯 **운영 팁**: 사용량은 `response.usage` 에 항상 포함. DB·CSV 에 기록해 두면 월말에 청구서와 대조 가능.

PART G

PART G
에러 처리 & 재시도

401 · 429 · 500 — 약 5분

에러 → 원인 → 대응

코드	예외 클래스	원인	대응
401	`AuthenticationError`	키 누락 / 만료 / 잘못된 키	`.env` 확인, 키 재발급
429	`RateLimitError`	초당 호출 / 토큰 한도 초과	exponential backoff (아래)
400	`BadRequestError`	잘못된 파라미터 (예: 존재하지 않는 모델)	요청 본문 점검
500/503	`APIError` / `APIConnectionError`	서버 일시 장애	재시도 (백오프)
insufficient_quota	`OpenAI 결제 잔액 부족`	무료 크레딧 소진	결제수단 등록 + spend limit

안전한 호출 패턴

```
import time, random
from openai import OpenAI, RateLimitError, APIError

client = OpenAI()

def ask_with_retry(msg: str, max_retries: int = 5) → str:
    delay = 1.0
    for attempt in range(max_retries):
        try:
            r = client.chat.completions.create(
                model='gpt-4o-mini',
                messages=[{'role':'user','content':msg}],
                max_tokens=300,
            )
            return r.choices[0].message.content
        except (RateLimitError, APIError) as e:
            if attempt == max_retries - 1:
                raise
            sleep_s = delay + random.uniform(0, 0.5)
            print(f"⚠️ {type(e).__name__} attempt {attempt+1}, sleep {sleep_s:.1f}s")
            time.sleep(sleep_s)
            delay *= 2   # 1s → 2s → 4s → 8s → 16s
```

> 💡 운영 환경에서는 `tenacity` 라이브러리로 더 깔끔하게.

PART H

PART H
미니 실습

> 직접 입력해 호출 — 약 15분

실습 과제 (15분)

1단계 — 대상 도메인 질문 3개 준비

- 예: "오라클 SQL 에서 ROWNUM 의 함정 3가지"
- 예: "TCP 3-way handshake 를 30단어로"
- 예: "한국어 텍스트 토큰화 시 BPE 의 한계"

3단계 — 비교 후 답변할 것

- 두 모델 응답의 품질 차이가 가격 차이를 정당화하는가?
- 어느 질문이 mini 로 충분, 어느 질문이 4o 가 필요했는가?

> 📌 결과를 CSV 로 남겨두면 다음 회차에서 비교용 데이터로 활용 가능.

2단계 — 두 모델 비교

```
for model in ['gpt-4o-mini', 'gpt-4o']:
    for q in questions:
        r = client.chat.completions.create(
            model=model,
            messages=[{'role':'user','content':q}],
            max_tokens=200,
            temperature=0.3,
        )
        cost = (r.usage.prompt_tokens * P_IN[model] +
                r.usage.completion_tokens * P_OUT[model]) / 1_000_000
        print(f"[{model}] tokens={r.usage.total_tokens} cost=${cost:.5f}")
        print(r.choices[0].message.content)
        print('-'*40)
```

핵심 8가지

1. ✅ LLM API = **HTTP req/resp** — 본질은 단순
2. ✅ API 는 **stateless** — 매 호출 독립, 대화 연속성 없음
3. ✅ "대화" 는 **클라이언트가 messages 에 과거를 누적**해 시뮬
4. ✅ **토큰** = 비용 단위 — tiktoken 으로 호출 전 추정
5. ✅ **REST · SDK old · SDK new** — 본질은 같은 HTTP 호출
6. ✅ 주요 파라미터 4: **temperature · top_p · max_tokens · penalties**
7. ✅ `system / user / assistant` 메시지 역할
8. ✅ 에러 5종 + **exponential backoff** 패턴

실습 과제 — 직접 해보기

필수

- [ ] `1.openai/1.intro/` 안의 **3개 코드** 모두 실행 (`1.restapi.py`, `2.sdk_old.py`, `3.sdk_new.py`)
- [ ] 도메인 질문 3개로 `gpt-4o-mini` vs `gpt-4o` 비교 + 비용 측정

옵션

- [ ] **Anthropic Console** 가입 + API Key 발급 (Agent 실습용)
- [ ] **tiktoken** 으로 평소 쓰는 프롬프트의 토큰 수 측정 — 한 줄 분석

📤 정리

- 호출 결과 + 토큰 사용량 + 비용 — Slack/메일 한 줄