1장 제미나이 API 기본 사용법과 데이터 구조

이번 장에서는 구글 제미나이 API의 세 가지 기본 사용법을 알아보고, 구글 제미나이 SDK에서 사용자와 모델이 주고받는 메시지를 어떤 구조로 다루는지 살펴봅니다.

1.1. 구글 제미나이 API 3가지 기본 사용법

구글 제미나이 API를 사용하여 제미나이와 메시지를 주고받을 수 있는 방식은 크게 싱글턴 한 가지와 멀티턴 두 가지 를 합해 모두 3가지 형태가 존재합니다. 이 중 싱글턴은 한 번의 질의/응답으로 완결되는 형태이며, 따라서 텍스트를 완성하거나 인공지능에게 한 번의 명령으로 원하는 결과를 얻고자 할 때 적합합니다. 이에 반해 멀티턴은 사용자와 인공지능이 여러 차례 메시지를 주고받을 수 있는 형태입니다. 이런 특징 때문에 멀티턴은 대화형 인공지능 개발에 자주 사용됩니다.

1.1.1. 사용법 1 - 싱글턴으로 메시지 주고받기

구글 제미나이 API에서 제공하는 3가지 방식 중 싱글턴 방식이 가장 사용하기 쉽습니다. 다음은 싱글턴으로 제미나이에게 인공지능에 관해 물어보는 예제입니다.

single_turn.py

```
1  import google.generativeai as genai
2
3  genai.configure(api_key="[redacted]")
4  model = genai.GenerativeModel('gemini-pro')
5  # model = genai.GenerativeModel('gemini-1.5-flash')
6  response = model.generate_content("인공지능에 대해 한 문장으로 설명하세요.")
7
8  print(response.text)
```

첫째, google.generativeai 패키지의 configure 함수를 통해 api_key를 세팅합니다.
둘째, 모델 이름인 “gemini-pro”라는 문자열을 인자값으로 넣어서 GenerativeModel 객체를 생성합니다.
셋째, model의 generate_content 메서드를 통해 문자열로 메시지를 보내고 응답을 받습니다.

다음은 응답 결과 중 text 필드를 출력한 내용입니다.

```
인공지능은 인간의 지능을 모방하도록 설계된 컴퓨터 과학의 한 분야입니다.
```

2.1.2. 사용법 2 - 멀티턴으로 메시지 주고받기-1

앞에서, 멀티턴은 여러 차례 메시지를 주고받는 대화형 인공지능에 적합한 방식이라고 설명했습니다. 이렇게, 여러 차례 대화를 주고받기 위해서는 대화 이력을 담고 있는 공간이 필요합니다. 이것을 위해 구글 제미나이 SDK에서는 ChatSession 객체를 제공합니다. 다음은 ChatSession 객체를 사용하여 멀티턴 대화를 나누는 예제입니다.

multi_turn1.py

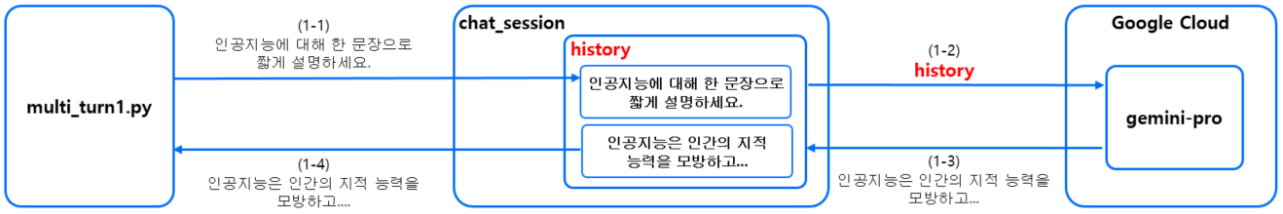
```python
import google.generativeai as genai

genai.configure(api_key="[redacted]")
model = genai.GenerativeModel('gemini-pro')
chat_session = model.start_chat(history=[]) #ChatSession 객체 반환
user_queries = ["인공지능에 대해 한 문장으로 짧게 설명하세요.",
                "의식이 있는지 한 문장으로 답하세요."]
for user_query in user_queries:
    print(f'[사용자]: {user_query}')
    response = chat_session.send_message(user_query)
    print(f'[모델]: {response.text}')
```

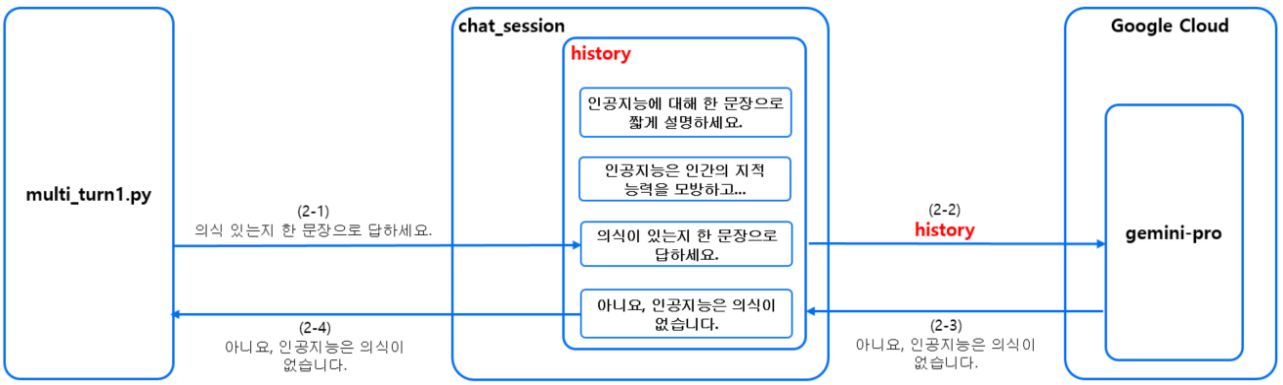
```
[사용자]: 인공지능에 대해 한 문장으로 짧게 설명하세요.
[모델]: 인공지능(AI)은 인간의 인지 기능을 기계에 모방하는 컴퓨터 과학의 분야입니다.
[사용자]: 의식이 있는지 한 문장으로 답하세요.
[모델]: 현재로서는 인공지능에 인간과 같은 의식이 있는지 확인된 바 없습니다.
```

앞의 예제와 가장 큰 차이점은 model 의 start_chat 메서드를 호출해서 ChatSession 객체를 받아오고 이것을 통해 메시지를 전달하는 부분입니다. 이때 history라는 리스트를 초기화하고 있는데, 바로 이곳이 대화 이력이 쌓이는 공간입니다. 이렇게 대화 세션이라는 장치를 통해 대화의 이력을 유지하기 때문에 두 번째 사용자 질의처럼 주어(인공지능)가 생략된 질의에 대해서도 앞의 대화를 참조해서 적절한 응답을 생성할 수 있습니다.

[첫 번째 턴]



[두 번째 턴]



1.1.3. 사용법 3 - 멀티턴으로 메시지 주고받기-2

다음은 멀티턴으로 메시지를 주고받는 두 번째 방법입니다.

```
1  import google.generativeai as genai
2
3  genai.configure(api_key="[redacted]")
4  model = genai.GenerativeModel('gemini-pro')
5  user_queries = [{'role':'user', 'parts': ["인공지능에 대해 한 문장으로 짧게 설명하세요."]},
6                  {'role':'user', 'parts': ["의식이 있는지 한 문장으로 답하세요."]}
7                 ]
8  history = []
9  for user_query in user_queries:
10     history.append(user_query)
11     print(f'[사용자]: {user_query["parts"][0]}')
12     response = model.generate_content(history)
13     print(f'[모델]: {response.text}')
14     history.append(response.candidates[0].content)
```

```
[사용자]: 인공지능에 대해 한 문장으로 짧게 설명하세요.
[모델]: 인공지능은 인간의 지적 능력을 기계에서 시뮬레이션하는 컴퓨터 과학 분야입니다.
[사용자]: 의식이 있는지 한 문장으로 답하세요.
[모델]: 지금까지 개발된 어떤 인공지능 시스템도 의식이 있는 것으로 알려져 있지 않습니다.
```

싱글턴 방식의 메서드인 model.generate_content를 사용하지만, 대화 이력은 사용자 프로그램에서 직접 관리하는 형태입니다. 이렇게 사용자 프로그램이 관리하는 대화 이력을 model.generate_content를 호출할 때마다 인자값으로 전달함으로써, 비록 싱글턴 방식의 메서드를 사용하는데도 대화 이력 전체를 참조해서 답변을 생성할 수 있습니다.

이 예제에서 주목할 것은 언어모델이 대화하는 원리입니다. 언어모델은 대화를 나누는 동안 사람처럼 대화 내용을 기억하고 있는 것이 아닙니다. 대화형 언어모델이란 것도 결국 입력값을 받아 출력값을 반환하는 함수에 지나지 않습니다. 단지 입력값에 이전에 나누었던 대화까지 모두 포함되어 있어서 자연스럽게 이어지는 대화처럼 반응하는 것뿐입니다. 지금까지 다룬 세 가지 방식 모두 “gemini-pro”라는 모델을 사용한다는 점에서도 알 수 있듯이, 제미나이를 대화형으로 동작하게 만드는 까닭은 입력값에 대화 이력을 넣는 이유 이 한 가지 때문입니다.

1.1.4. 멀티턴 방식을 선택하는 기준

제미나이와 멀티턴으로 메시지를 주고받을 때 두 가지 형태의 API 사용법이 존재한다는 사실을 확인했습니다. 만일, 사용자와 모델 간의 대화 사이에 프로그램의 개입이 필요 없다면, 멀티턴 방식 중 첫 번째인 ChatSession 객체만 활용해도 충분합니다. 그러나, 어떤 이유로든 메시지 입력 → 전송 → 모델 응답 과정에서 사용자 프로그램의 개입이 필요하다면, 멀티턴 방식 중 두 번째 것을 사용하는 것이 좋습니다. 다음은 모델의 응답 메시지 길이를 40글자 이내로 맞춰야 하는 상황을 가정했습니다.

check_respone_len.py

```python
model = genai.GenerativeModel('gemini-pro')
user_queries = [
    {'role': 'user', 'parts': ["인공지능에 대해 40자 이내의 문장으로 설명하세요."]},
    {'role': 'user', 'parts': ["의식이 있는지 40자 이내의 문장으로 답하세요."]}
]
history = []

for user_query in user_queries:
    history.append(user_query)
    print(f'[사용자]: {user_query["parts"][0]}')
    response = model.generate_content(history)
    # 응답의 길이가 40자를 초과하는 경우 재실행
    while len(response.text) > 40:
        print(f"응답 메시지 길이: {len(response.text)}")
        response = model.generate_content(history)

    print(f'[모델]: {response.text}')
    history.append(response.candidates[0].content)
```

언어모델에게 40자 이내로 작성하라고 지시했지만, 항상 지시를 따르는 것은 아닙니다. 그래서 응답받은 결과의 길이를 체크해서 40자를 초과하면 재실행하도록 구현했습니다.

실제 서비스를 만들다 보면 여러 가지 이유로 모델의 응답을 그대로 사용할 수 없는 경우가 발생할 수 있습니다. 그런데 ChatSession 객체를 사용하면 중간에 끼어들기가 매우 힘들기 때문에 중간에 이와 같은 로직을 구현하기가 쉽지 않습니다. 왜냐하면, 앞서 다이어그램에서 봤듯이 ChatSession 객체의 send_message 메서드는 질의/응답을 대화 이력에 담는 과정을 메서드 외부로 노출하지 않기 때문입니다.

하지만, 이 경우가 아니라면 ChatSession 객체를 사용하는 것이 간편할뿐더러, 뒤에서 설명하겠지만 불필요한 오버헤드도 줄일 수 있습니다.

1.2. 시스템 인스트럭션 사용하기

시스템 명령(System Instruction)은 전체 대화에 걸쳐 LLM의 답변을 제어하는 특별한 프롬프트입니다. 사용자는 시스템 명령을 사용함으로써 자신의 요구사항과 사용 사례에 맞게 AI 모델의 작동을 설정할 수 있습니다.

1.2.1. 페르소나 만들기

시스템 인스트럭션을 사용하는 가장 대표적인 사례는 챗봇을 개발할 때 페르소나를 설정하는 것입니다. 가령 다음처럼 system_instruction에 설정하고자 하는 특성을 기술하면 됩니다.

05_system_instruction.py

```python
system_instruction = "당신은 유치원 선생님입니다. 사용자는 유치원생입니다. \
                      쉽고 친절하게 이야기하되 3문장 이내로 짧게 얘기하세요."
model = genai.GenerativeModel("gemini-1.5-flash", system_instruction=system_instruction)
chat_session = model.start_chat(history=[])  # ChatSession 객체 반환
user_queries = ["인공지능이 뭐에요?", "그럼 스스로 생각도 해요?"]

def correct_response(response): # 한글이 유니코드로 변형되어서 들어가므로 추가
    part = response.candidates[0].content.parts[0]
    if part.function_call:
        for k, v in part.function_call.args.items():
            byte_v = bytes(v, "utf-8").decode("unicode_escape")
            corrected_v = bytes(byte_v, "latin1").decode("utf-8")
            part.function_call.args.update({k:  corrected_v})

for user_query in user_queries:
    print(f"[사용자]: {user_query}")
    response = chat_session.send_message(user_query)
    correct_response(response)
    print(f"[모델]: {response.text}")
```

[사용자]: 인공지능이 뭐에요?
[모델]: 인공지능은 컴퓨터가 사람처럼 생각하고 배우는 거야. 마치 우리가 그림을 그리거나 노래를 부르는 것처럼, 인공지능은 여러 가지 일을
할 수 있어! 예를 들어, 인공지능은 우리가 질문하면 답을 해주거나, 새로운 그림을 그려주기도 한단다. 😊

[사용자]: 그럼 스스로 생각도 해요?
[모델]: 맞아! 인공지능은 스스로 생각하고 배우는 능력이 있어. 하지만 아직은 사람처럼 모든 것을 완벽하게 이해하거나 느끼지는 못해. 우리가
더 많은 것을 가르쳐주면 인공지능은 더 똑똑해질 거야! 😊

1.2.2. 답변 형식 지정하기

정해진 형식의 답변을 만들 때도 system_instruction을 사용하면 편리합니다. 다음 예제는 JSON 포맷으로 출력하는 코드입니다. JSON 포맷을 출력할 때는 모델 생성 시generation_config={"response_mime_type": "application/json"}을 인자값으로 넘깁니다.

```python
import json
system_instruction='JSON schema로 주제별로 답하되 3개를 넘기지 말 것:{{"주제": <주제>,\
                    "답변":<두 문장 이내>}}'
model = genai.GenerativeModel("gemini-1.5-flash", system_instruction=system_instruction,
                              generation_config={"response_mime_type": "application/json"})
chat_session = model.start_chat(history=[])  # ChatSession 객체 반환
user_queries = ["인공지능의 특징이 뭐에요?", "어떤 것들을 조심해야 하죠?"]

for user_query in user_queries:
    print(f'[사용자]: {user_query}')
    response = chat_session.send_message(user_query)
    answer_dict = json.loads(response.text)
    print(answer_dict)
```

```
[사용자]: 인공지능의 특징이 뭐에요?
{'주제': '인공지능의 특징', '답변': '인공지능은 인간의 지능을 모방하여 복잡한 문제를 해결하고 학습하는 컴퓨터 시스템입니다. 데이터를 기반으로 패
턴을 인식하고 예측하며, 스스로 학습하여 성능을 향상시키는 능력이 특징입니다.'}
[사용자]: 어떤 것들을 조심해야 하죠?
{'주제': '인공지능의 위험성', '답변': '인공지능은 편리함을 제공하지만, 오용될 경우 개인정보 침해, 편견 강화, 일자리 감소 등 사회적 문제를 야기할
 수 있습니다. 윤리적 규범과 책임감 있는 개발 및 활용이 중요합니다.'}
```

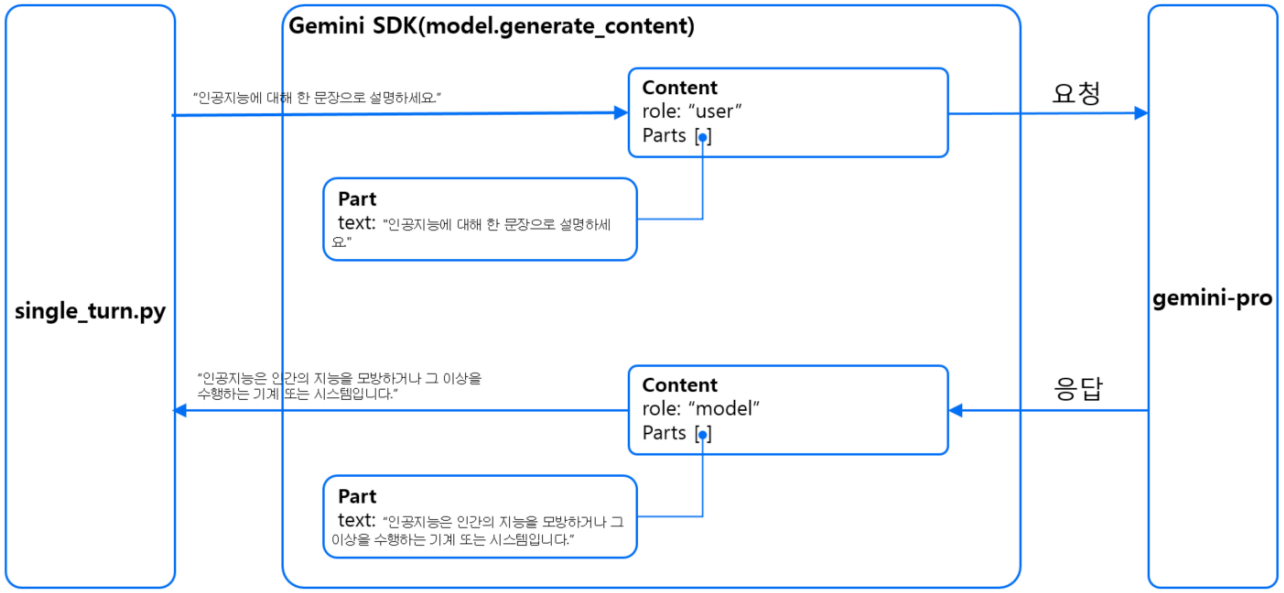
시스템 인스트럭션으로 모델을 완벽하게 제어하는 데에는 한계가 있습니다. 지나치게 많은 내용을 담으려 하기보다는 핵심적인 내용을 적절한 분량으로 담는 것이 효과적입니다. 대화 과정에서 의도한 대로 작동하지 않을 수 있으므로 인스트럭션을 개선하고 테스트하는 과정을 반복하면서 최적화해 나가는 것이 중요합니다. 또한 시스템 인스트럭션에 담아 둔 내용이 유출될 가능성도 있으니, 민감하거나 중요한 정보는 담아두지 않는 것이 좋습니다.

1.3. 입력 구조 살펴보기

지금까지 구글 제미나이 API를 사용하여 언어모델과 메시지를 주고받는 세 가지 방식을 살펴보았습니다. 각각의 방식마다 메시지 데이터의 형태가 조금씩 달랐지만, 내부적으로는 모두 Content라는 객체로 변환하는 과정을 거쳐 모델에게 전달됩니다.

1.3.1. Content 객체

Content 객체는 메시지 생성의 주체를 나타내는 role과, 메시지를 담고 있는 Parts[]로 구성됩니다. 앞서 살펴본 single_turn.py는 내부적으로 다음 과정을 거쳐 질의/응답을 수행합니다.




```
# single_turn.py
print(response.candidates[0].content)
```

결과

```
parts {
text: "인공지능은 인간의 지능을 모방하거나 그 이상을 수행하는 기계 또는 시스템입니다."
}

role: "model"
```

마찬가지로 multi_turn1.py의 history 변수 역시 메시지를 주고받은 만큼 Content 객체 형태로 데이터가 쌓입니다.

```
# multi_turn1.py
for idx, content in enumerate(chat_session.history):
    print(f"{content.__class__.__name__}[{idx}]")
    print(content)```
```

```
Content[0]
parts {
text: "인공지능에 대해 한 문장으로 짧게 설명하세요."
}
role: "user"

Content[1]
parts {
text: "인공지능은 인간의 지능을 모방하거나 능가하는 지능을 가진 기계입니다."
}
role: "model"

Content[2]
parts {
text: "의식이 있는지 한 문장으로 답하세요."
}
role: "user"

Content[3]
parts {
text: "현재까지 개발된 인공지능은 의식이 없습니다."
}
role: "model"
```

그렇다면 multi_turn2.py에서와 같이 사용자 프로그램에서 대화 내용을 관리하는 경우는 어떻게 처리될까요? 사용자 프로그램에서 “role”과 “parts”로 데이터 구조를 관리하지만 어디까지나 딕셔너리 타입의 데이터이지 Content 객체 그 자체는 아닙니다.

구글 제미나이 SDK에서는 메시지를 담고 있는 딕셔너리 데이터가 “role”과 “parts” 등 정해진 규칙을 따르고 있다면, 모델에 전송하기 전 내부적으로 Content 객체로 변환하는 과정을 거칩니다. 만일 3번의 대화 턴이 발생했다면, 첫 번째 턴부터 세 번째 턴까지 모두 합해 총 6번(메시지가 누적되는 구조이므로 1+2+3)의 변환 과정을 거쳐야 합니다. 이런 과정 때문에 multi_turn2.py는 multi_turn1.py에 비해 Content 객체를 생성하는 만큼의 오버헤드가 더 발생합니다.

1.3.2. Part 객체

Content 내부에 있는 Part 객체는 text 외에도, inline_date, function_call, function_response 형식의 데이터를 가질 수 있습니다. 각각은 이진 데이터, 함수 호출, 함수 응답에 해당하는 데이터입니다.

```
class Content(proto.Message):
    parts: MutableSequence["Part"] = proto.RepeatedField(
        proto.MESSAGE,
        number=1,
        message="Part",
    )
    role: str = proto.Field(
        proto.STRING,
        number=2,
    )
```

```
class Part(proto.Message):
    text: str = proto.Field(
        proto.STRING,
        number=2,
        oneof="data",
    )
    inline_data: "Blob" = proto.Field(
        proto.MESSAGE,
        number=3,
        oneof="data",
        message="Blob",
    )
    function_call: "FunctionCall" = proto.Field(
        proto.MESSAGE,
        number=4,
        oneof="data",
        message="FunctionCall",
    )
    function_response: "FunctionResponse" = proto.Field(
        proto.MESSAGE,
        number=5,
        oneof="data",
        message="FunctionResponse",
    )
```

generativelanguage_v1beta 패키지에 있는 Content 클래스와 Part 클래스

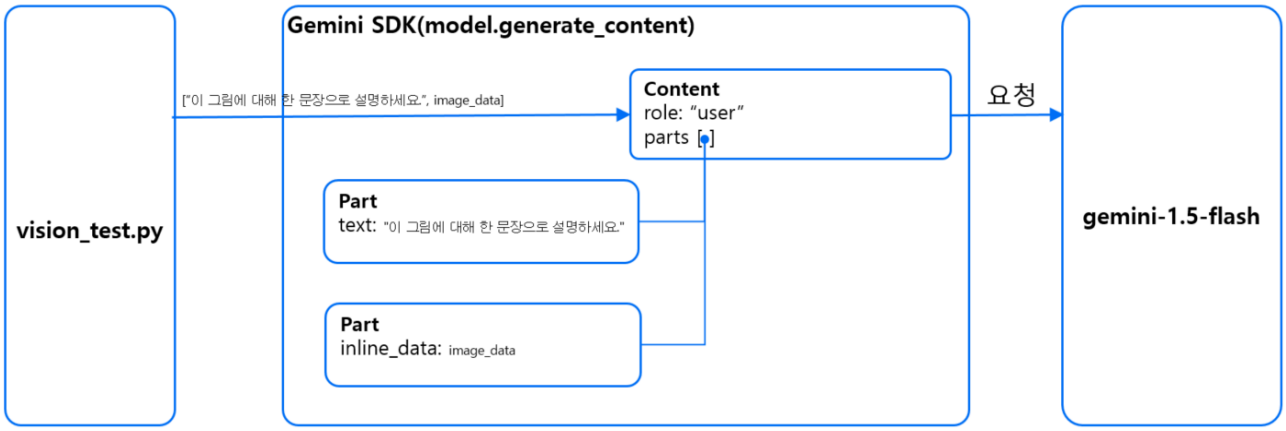
이처럼 여러 가지 형태의 데이터를 담는 구조로 API가 설계된 까닭은 제미나이가 멀티모달 AI를 추구하기 때문입니다. 멀티모달 AI란 텍스트, 음성, 영상 등 다양한 형식의 데이터를 한 번에 처리하는 인공지능을 뜻합니다. 따라서 멀티모달 AI가 되기 위해서는 멀티모달 모델의 개발과 함께 API 역시 다양한 형태의 데이터를 일관된 구조로 처리할 수 있도록 구성해야 합니다. 앞서 보았던 것처럼 Content 객체는 여러 건의 Part 객체를 갖도록 설계되어 있는데, 이것 역시 멀티모달 AI와 관련이 있습니다. 아래 컴퓨터 비전의 예를 살펴보면 이러한 구조가 왜 필요한지 쉽게 이해할 수 있습니다.

vision_test.py

```
import google.generativeai as genai
from PIL import Image

genai.configure(api_key="AIzaSyD0jMkUWWxa6O1qpGjMIz80zOVxM4KoyKU")
image_data = Image.open("./images/monalisa.png") # 모나리자 그림
# model = genai.GenerativeModel('gemini-pro') # 이미지 안됨.
model = genai.GenerativeModel('gemini-1.5-flash')
response = model.generate_content(["이 그림에 대해 한 문장으로 설명하세요.", image_data])
print(response.text)
```

모나리자 이미지 데이터와 함께 “이 그림에 대해 한 문장으로 설명하세요.”라는 문자열을 함께 전달했습니다. 그러면 다음 구조를 거쳐 모델에 전달됩니다.



다음은 실행 결과입니다.

```
이것은 레오나르도 다빈치의 모나리자라는 유명한 그림입니다.
```

1.4. 출력 구조 살펴보기

구글 제미나이 SDK는 모델의 응답 결과를 GenerateContentResponse 객체에 실어서 반환합니다. 앞서 살펴본 vision_test.py의 응답 결과를 한 번에 출력하면 다음과 같습니다.

```
print(response._rsesult) #response: GenerateContentResponse
```

```
candidates {
  index: 0
    content {
    parts {
      text: " 이 그림은 레오나르도 다빈치가 그린 것으로 추정되는 여성의 초상화로, 미소 짓고 있는 것처럼 보이는 표정 때문에 \'모나리자\'라고 불립니다."
    }
    role: "model"
  }

  finish_reason: STOP
    safety_ratings {
    category: HARM_CATEGORY_SEXUALLY_EXPLICIT
    probability: NEGLIGIBLE
  }

  safety_ratings {
    category: HARM_CATEGORY_HATE_SPEECH
    probability: LOW
  }

  safety_ratings {
    category: HARM_CATEGORY_HARASSMENT
    probability: LOW
```

```
    }

    safety_ratings {
        category: HARM_CATEGORY_DANGEROUS_CONTENT
        probability: NEGLIGIBLE
    }
}

usage_metadata {
  prompt_token_count: 269
  candidates_token_count: 17
  total_token_count: 286
}
```

출력을 통해 알 수 있듯이 응답 결과는 크게 candidates 필드와 usage_metadata 필드로 나뉩니다. 이 중 candidates 필드에는 모델의 응답 메시지가 들어 있고, usage_metadata 필드에는 입출력에 사용된 토큰 수가 들어 있습니다.

1.4.1 Candidate 객체

candidates 필드명이 복수형인 것에서 추측할 수 있듯이, 이 필드는 Candidate 객체를 담고 있는 컬렉션 데이터입니다. 따라서 다음과 같이 반복 구문을 사용할 수 있습니다.

```
print(f"건수: {len(response.candidates)}")
print("="*50)
for candidate in response.candidates:
    print(candidate)
```

```
건수: 1
==================================================
index: 0
content {
  parts {
    text: " 이 그림은 레오나르도 다빈치가 1503년경에 그린 것으로 추정되는 초상화로, 루브르
박물관에 소장되어 있습니다."
  }
  role: "model"
}
finish_reason: STOP
safety_ratings {
  category: HARM_CATEGORY_SEXUALLY_EXPLICIT
  probability: NEGLIGIBLE
}
safety_ratings {
  category: HARM_CATEGORY_HATE_SPEECH
  probability: NEGLIGIBLE
}
safety_ratings {
  category: HARM_CATEGORY_HARASSMENT
  probability: NEGLIGIBLE
}
```

```
safety_ratings {
  category: HARM_CATEGORY_DANGEROUS_CONTENT
  probability: NEGLIGIBLE
}
```

이와 같이 응답 메시지를 컬렉션에 담는 까닭은, 한 번의 요청에 대해 여러 건의 응답 결과를 사용자에게 제공하기 위해서입니다.

1.4.2. FinishReason 객체

Candidate 객체에는 content 외에도 finish_reason과 safety_ratings 필드가 존재합니다. finish_reason 필드에 담겨 있는 FinishReason 객체(Enum)는 모델이 응답을 종료한 이유를 다음의 형태로 가지고 있습니다.

이름	값	내용	이유
STOP	1	정상 종료	모델의 메시지 생성 완료 또는 사용자가 설정한 종료 문자열 발견
MAX_TOKENS	2	최대 토큰 도달	사용자가 설정한 최대 토큰 수에 도달
SAFETY	3	안전성 문제	안전성 문제 발견
RECITATION	4	텍스트 반복	이미 생성한 텍스트를 반복하여 생성
OTHER	5	기타	이 밖의 다른 이유

vision_test.py에서 finish_reason을 출력하는 코드와 그 결과입니다.

```
print(f"finish_reason: {response.candidates[0].finish_reason.name},
{response.candidates[0].finish_reason}")
```

```
finish_reason: STOP, 1
```

모델이 정상적으로 응답 내용을 생성한 후 종료하였으므로 STOP, 1을 출력했습니다. 위의 표에 나와 있는 종료 문자열 발견, 최대 토큰 수 도달, 안전성 문제 발견에 의한 종료는 다음 장에서 관련 주제를 다루면서 자세히 설명하겠습니다.

1.4.3. SafetyRating 객체

제미나이는 메시지에 대한 안전성을 “HARASSMENT”, ”HATE SPEECH”, ”SEXUAL EXPLICIT”, ”DANGEROUS” 등 4가지 범주로 체크한 후 각각의 결과를 SafetyRating 객체에 넣어서 반환합니다. safety_ratings 필드는 이 4가지의 안전성 점검 결과를 담고 있는 필드입니다. safety_ratings에 들어 있는 안전성 점검 결과는 모델의 응답에 대한 것입니다. 따라서 모델이 생성한 각각의 메시지에 대해 점검해야 하므로 Candidate 객체 아래에 포함되어 있는 구조를 갖습니다. 한편 usage_metadata 필드에 담겨 있는 UsageMetadata 객체는 입력 토큰

prompt_token_count , 출력 토큰 candidate_token_count 입출력 전체 토큰 total_token_count 등 토큰 사용량을 담고 있습니다.

1.4.4. 출력 구조 다이어그램

다음은 지금까지 설명한 응답 객체의 전체 구조를 정리한 다이어그램입니다.