In [ ]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Gemini API: Quickstart with Python

View on Google AI (https://ai.google.dev/tutorials/python_quickstart)

Run in Google Colab (https://colab.research.google.com/github/google/generative-ai-docs/blob/main/site/en/tutorials/python_quickstart.ipynb)

This quickstart demonstrates how to use the Python SDK for the Gemini API, which gives you access to Google's Gemini large language models. In this quickstart, you will learn how to:

1. Set up your development environment and API access to use Gemini.
2. Generate text responses from text inputs.
3. Generate text responses from multimodal inputs (text and images).
4. Use Gemini for multi-turn conversations (chat).
5. Use embeddings for large language models.

Prerequisites

You can run this quickstart in Google Colab (https://colab.research.google.com/github/google/generative-ai-docs/blob/main/site/en/tutorials/python_quickstart.ipynb), which runs this notebook directly in the browser and does not require additional environment configuration.

Alternatively, to complete this quickstart locally, ensure that your development environment meets the following requirements:

- Python 3.9+
- An installation of `jupyter` to run the notebook.

Setup

Install the Python SDK

The Python SDK for the Gemini API, is contained in the `google-generativeai` (https://pypi.org/project/google-generativeai/) package. Install the dependency using pip:

In [ ]:
```
!pip install -q -U google-generativeai
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 142.1/142.1 kB 2.0 MB/s eta 0:0
0:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 663.6/663.6 kB 17.7 MB/s eta 0:
00:00
```

Import packages

Import the necessary packages.

In [ ]:
```
import pathlib
import textwrap

import google.generativeai as genai

from IPython.display import display
from IPython.display import Markdown


def to_markdown(text):
  text = text.replace('•', '  *')
  return Markdown(textwrap.indent(text, '> ', predicate=lambda _: True))
```

In [ ]:
```
# Used to securely store your API key
from google.colab import userdata
```

Setup your API key

Before you can use the Gemini API, you must first obtain an API key. If you don't already have one, create a key with one click in Google AI Studio.

Get an API key (https://makersuite.google.com/app/apikey)

In Colab, add the key to the secrets manager under the "🔑" in the left panel. Give it the name `GOOGLE_API_KEY` .

Once you have the API key, pass it to the SDK. You can do this in two ways:

- Put the key in the `GOOGLE_API_KEY` environment variable (the SDK will automatically pick it up from there).
- Pass the key to `genai.configure(api_key=...)`

In [ ]:
```
# Or use `os.getenv('GOOGLE_API_KEY')` to fetch an environment variable.
GOOGLE_API_KEY=userdata.get('GOOGLE_API_KEY')

genai.configure(api_key=GOOGLE_API_KEY)
```

List models

Now you're ready to call the Gemini API. Use `list_models` to see the available Gemini models:

- `gemini-pro` : optimized for text-only prompts.
- `gemini-pro-vision` : optimized for text-and-images prompts.

In [ ]:
```
for m in genai.list_models():
  if 'generateContent' in m.supported_generation_methods:
    print(m.name)
```

```
models/gemini-1.0-pro
models/gemini-1.0-pro-001
models/gemini-1.0-pro-latest
models/gemini-1.0-pro-vision-latest
models/gemini-1.5-pro-latest
models/gemini-pro
models/gemini-pro-vision
```

Note: For detailed information about the available models, including their capabilities and rate limits, see Gemini models (https://ai.google.dev/models/gemini). There are options for requesting rate limit increases (https://ai.google.dev/docs/increase_quota). The rate limit for Gemini-Pro models is 60 requests per minute (RPM).

The `genai` package also supports the PaLM family of models, but only the Gemini models support the generic, multimodal capabilities of the `generateContent` method.

Generate text from text inputs

For text-only prompts, use the `gemini-pro` model:

In [ ]:
```
model = genai.GenerativeModel('gemini-1.5-pro-latest')
```

The `generate_content` method can handle a wide variety of use cases, including multi-turn chat and multimodal input, depending on what the underlying model supports. The available models only support text and images as input, and text as output.

In the simplest case, you can pass a prompt string to the `GenerativeModel.generate_content`

In [ ]:
```
%%time
response = model.generate_content("What is the meaning of life?")
```

```
CPU times: user 211 ms, sys: 27.4 ms, total: 238 ms
Wall time: 13.7 s
```

In simple cases, the `response.text` accessor is all you need. To display formatted Markdown text, use the `to_markdown` function:

In [ ]:
```
to_markdown(response.text)
```

Out[8]:
```
<IPython.core.display.Markdown object>
```

If the API failed to return a result, use `GenerateContentResponse.prompt_feedback` to see if it was blocked due to safety concerns regarding the prompt.

In [ ]:
```
response.prompt_feedback
```

Out[9]:

Gemini can generate multiple possible responses for a single prompt. These possible responses are called `candidates`, and you can review them to select the most suitable one as the response.

View the response candidates with `GenerateContentResponse.candidates` (https://ai.google.dev/api/python/google/ai/generativelanguage/GenerateContentResponse#candi

In [ ]:
```
1 response.candidates
```

Out[10]:
```
[content {
  parts {
    text: "The meaning of life is a complex and multifaceted question that ha
s been pondered by philosophers and theologians for centuries. There is no on
e definitive answer, as the meaning of life is subjective and can vary greatl
y from person to person. \n\nHere are a few perspectives on the meaning of li
fe:\n\n* **Philosophical perspectives:**\n    * **Nihilism:** This philosophy
suggests that life has no intrinsic meaning or purpose. \n    * **Existential
ism:** This philosophy emphasizes individual existence, freedom, and choice.
It suggests that individuals create their own meaning in life through their a
ctions and choices.\n    * **Humanism:** This philosophy emphasizes the value
and agency of human beings. It suggests that the meaning of life is found in
living a good life and contributing to the well-being of others.\n* **Religio
us perspectives:**\n    * Many religions believe that the meaning of life is
to serve and worship a higher power.\n    * Some religions believe in an afte
rlife and that the meaning of life is to prepare for this next stage of exist
ence.\n* **Personal perspectives:**\n    * **Happiness:** Many people believe
that the meaning of life is to find happiness and fulfillment.\n    * **Purpo
se:** Others believe that the meaning of life is to find and fulfill a specif
ic purpose. \n    * **Connection:** Some people find meaning in their relatio
nships with others and in their contributions to society.\n\nUltimately, the
meaning of life is a personal journey that each individual must discover for
themselves. It is a question that can be explored through introspection, refl
ection, and engagement with the world around us. \n"
  }
  role: "model"
}
finish_reason: STOP
index: 0
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
}
]
```

By default, the model returns a response after completing the entire generation process. You can also stream the response as it is being generated, and the model will return chunks of the response as soon as they are generated.

To stream responses, use `GenerativeModel.generate_content(..., stream=True)` (https://ai.google.dev/api/python/google/generativeai/GenerativeModel#generate_content).

In [ ]:

```
%%time
response = model.generate_content("What is the meaning of life?", stream=T
```

```
CPU times: user 258 ms, sys: 26 ms, total: 284 ms
Wall time: 16.5 s
```

In [ ]:

```
for chunk in response:
  print(chunk.text)
  print("_"*80)
```

```
The
_____
___
 meaning of life is a complex and multifaceted question that has been pondere
d by philosophers and
_____
___
 theologians for centuries. There is no one definitive answer, as the meaning
of life
_____
___
 is subjective and can vary greatly from person to person.

**Some possible perspectives on the meaning of life include:**

* **Purpose and Goals:** Life may
_____
___
 have an inherent purpose or goal, whether it is determined by a higher powe
r, fate, or personal aspirations. Finding and fulfilling this purpose can pro
vide a sense of
_____
___
 meaning and direction.
* **Relationships and Connection:** Human connection and relationships with o
thers can bring joy, love, and a sense of belonging. Building meaningful conn
ections can be a significant source of life's meaning.
* **Experiences
_____
___
 and Growth:** Life is a journey of experiences, both positive and negative.
Through these experiences, we learn, grow, and evolve as individuals. Embraci
ng new experiences and challenges can contribute to a fulfilling life.
* **Contribution and Legacy
_____
___
:** Making a positive impact on the world, leaving a legacy, or contributing
to something larger than oneself can provide a sense of purpose and meaning.
* **Happiness and Well-being:** The pursuit of happiness and well-being is of
ten seen as a fundamental human goal. Finding joy in everyday life, cultivati
ng gratitude,
_____
___
 and experiencing positive emotions can enhance the meaning of life.
* **Spirituality and Transcendence:** For some, the meaning of life is found
in spiritual or religious beliefs, connecting with a higher power, or seeking
enlightenment.

**Ultimately, the meaning of life is a personal journey that each individual
must discover for themselves
_____
___
.** It is an ongoing process of exploration, reflection, and growth.

**Here are some tips for finding meaning in life:**

* **Reflect on your values and beliefs.**
```

```
* **Explore your interests and passions.**
* **Connect with others and build meaningful relationships.**
* **Set goals and pursue
```

```
 your aspirations.**
* **Contribute to your community or a cause you care about.**
* **Practice gratitude and mindfulness.**
* **Seek out new experiences and challenges.**

Remember, the meaning of life is not a destination but a journey. Embrace the
process of discovery and enjoy the ride.
```

When streaming, some response attributes are not available until you've iterated through all the response chunks. This is demonstrated below:

In [ ]:
```
response = model.generate_content("What is the meaning of life?", stream=T
```

The `prompt_feedback` attribute works:

In [ ]:
```
response.prompt_feedback
```

Out[14]:

But attributes like `text` do not:

In [ ]:
```
try:
  response.text
except Exception as e:
  print(f'{type(e).__name__}: {e}')
```

```
IncompleteIterationError: Please let the response complete iteration before a
ccessing the final accumulated
attributes (or call `response.resolve()`)
```

Generate text from image and text inputs

Gemini provides a multimodal model (`gemini-pro-vision`) that accepts both text and images and inputs. The `GenerativeModel.generate_content` API is designed to handle multimodal prompts and returns a text output.

Let's include an image:

In [ ]:
```
!curl -o image.jpg https://t0.gstatic.com/licensed-image?q=tbn:ANd9GcQ_Kev
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Curre
nt
                                 Dload  Upload   Total   Spent    Left  Speed
100  405k  100  405k    0     0  5195k      0 --:--:-- --:--:-- --:--:-- 5260
k
```

In [ ]:
```
import PIL.Image

img = PIL.Image.open('image.jpg')
img
```

Out[17]:

Use the `gemini-pro-vision` model and pass the image to the model with `generate_content`.

In [ ]:
```
model = genai.GenerativeModel('gemini-1.5-pro-latest')
```

In [ ]:
```
response = model.generate_content(img)

to_markdown(response.text)
```

Out[19]: `<IPython.core.display.Markdown object>`

To provide both text and images in a prompt, pass a list containing the strings and images:

In [ ]:
```
response = model.generate_content(["Write a short, engaging blog post base
response.resolve()
```

In [ ]:
```
to_markdown(response.text)
```

Out[21]: <IPython.core.display.Markdown object>

Chat conversations

Gemini enables you to have freeform conversations across multiple turns. The `ChatSession` class simplifies the process by managing the state of the conversation, so unlike with `generate_content`, you do not have to store the conversation history as a list.

Initialize the chat:

In [ ]:
```
model = genai.GenerativeModel('gemini-1.5-pro-latest')
chat = model.start_chat(history=[])
chat
```

Out[22]:
```
ChatSession(
    model=genai.GenerativeModel(
        model_name='models/gemini-1.5-pro-latest',
        generation_config={},
        safety_settings={},
        tools=None,
        system_instruction=None,
    ),
    history=[]
)
```

Note: The vision model `gemini-pro-vision` is not optimized for multi-turn chat.

The `ChatSession.send_message` method returns the same `GenerateContentResponse` type as `GenerativeModel.generate_content` (https://ai.google.dev/api/python/google/generativeai/GenerativeModel#generate_content). It also appends your message and the response to the chat history:

In [ ]:
```
response = chat.send_message("In one sentence, explain how a computer work
to_markdown(response.text)
```

Out[23]: <IPython.core.display.Markdown object>

In [ ]:
```
1 chat.history
```

Out[24]:
```
[parts {
   text: "In one sentence, explain how a computer works to a young child."
 }
 role: "user",
 parts {
   text: "Just like you use your brain to think, a computer uses its special
brain, called a CPU, to follow instructions and solve problems! \n"
 }
 role: "model"]
```

You can keep sending messages to continue the conversation. Use the `stream=True` argument to stream the chat:

In [ ]:
```
1 response = chat.send_message("Okay, how about a more detailed explanation
2
3 for chunk in response:
4   print(chunk.text)
5   print("_"*80)
```

```
Computers
________________________________________________________________________________
___
 operate through a fascinating interplay of hardware and software components.
At their core,
________________________________________________________________________________
___
 computers rely on the central processing unit (CPU), which acts as the brai
n,
________________________________________________________________________________
___
 executing instructions and performing calculations.  These instructions come
from software programs, which are sets of coded commands that tell the comput
er what to do.  The computer'
________________________________________________________________________________
___
s memory, including RAM and storage drives, plays a crucial role in holding b
oth data and instructions for the CPU to access. Input and output devices, su
ch as
________________________________________________________________________________
___
 keyboards, mice, and monitors, allow users to interact with the computer, pr
oviding data and receiving results.  All these components work together in a
coordinated manner, following the principles of logic and algorithms, to proc
ess information and accomplish tasks as instructed
________________________________________________________________________________
___
 by the user.

________________________________________________________________________________
___
```

`glm.Content` objects contain a list of `glm.Part` objects that each contain either a text (string) or inline_data (`glm.Blob`), where a blob contains binary data and a `mime_type`. The chat history is available as a list of `glm.Content` objects in `ChatSession.history`:

In [ ]:
```
for message in chat.history:
  display(to_markdown(f'**{message.role}**: {message.parts[0].text}'))
```

```
<IPython.core.display.Markdown object>

<IPython.core.display.Markdown object>

<IPython.core.display.Markdown object>

<IPython.core.display.Markdown object>
```

Count tokens

Large language models have a context window, and the context length is often measured in terms of the **number of tokens**. With the Gemini API, you can determine the number of tokens per any `glm.Content` object. In the simplest case, you can pass a query string to the `GenerativeModel.count_tokens` method as follows:

In [ ]:
```
model.count_tokens("What is the meaning of life?")
```

Out[27]:
```
total_tokens: 7
```

Similarly, you can check `token_count` for your `ChatSession`:

In [ ]:
```
model.count_tokens(chat.history)
```

Out[28]:
```
total_tokens: 212
```

Use embeddings

Embedding (https://developers.google.com/machine-learning/glossary#embedding-vector) is a technique used to represent information as a list of floating point numbers in an array. With Gemini, you can represent text (words, sentences, and blocks of text) in a vectorized form, making it easier to compare and contrast embeddings. For example, two texts that share a similar subject matter or sentiment should have similar embeddings, which can be identified through mathematical comparison techniques such as cosine similarity. For more on how and why you should use embeddings, refer to the Embeddings guide (https://ai.google.dev/docs/embeddings_guide).

Use the `embed_content` method to generate embeddings. The method handles embedding for the following tasks (`task_type`):

Task Type	Description
RETRIEVAL_QUERY	Specifies the given text is a query in a search/retrieval setting.

Task Type	Description
RETRIEVAL_DOCUMENT	Specifies the given text is a document in a search/retrieval setting. Using this task type requires a `title`.
SEMANTIC_SIMILARITY	Specifies the given text will be used for Semantic Textual Similarity (STS).
CLASSIFICATION	Specifies that the embeddings will be used for classification.
CLUSTERING	Specifies that the embeddings will be used for clustering.

In [ ]:
```
result = genai.embed_content(
    model="models/embedding-001",
    content="What is the meaning of life?",
    task_type="retrieval_document",
    title="Embedding of single string")

# 1 input > 1 vector output
print(str(result['embedding'])[:50], '... TRIMMED]')
```

```
[-0.003216741, -0.013358698, -0.017649598, -0.0091 ... TRIMMED]
```

Note: The `retrieval_document` task type is the only task that accepts a title.

To handle batches of strings, pass a list of strings in `content`:

In [ ]:
```
result = genai.embed_content(
    model="models/embedding-001",
    content=[
      'What is the meaning of life?',
      'How much wood would a woodchuck chuck?',
      'How does the brain work?'],
    task_type="retrieval_document",
    title="Embedding of list of strings")

# A list of inputs > A list of vectors output
for v in result['embedding']:
  print(str(v)[:50], '... TRIMMED ...')
```

```
[0.0040260437, 0.004124458, -0.014209415, -0.00183 ... TRIMMED ...
[-0.004049845, -0.0075574904, -0.0073463684, -0.03 ... TRIMMED ...
[0.025310587, -0.0080734305, -0.029902633, 0.01160 ... TRIMMED ...
```

While the `genai.embed_content` function accepts simple strings or lists of strings, it is actually built around the `glm.Content` type (like `GenerativeModel.generate_content` (https://ai.google.dev/api/python/google/generativeai/GenerativeModel#generate_content)). `glm.Content` objects are the primary units of conversation in the API.

While the `glm.Content` object is multimodal, the `embed_content` method only supports text embeddings. This design gives the API the *possibility* to expand to multimodal embeddings.

In [ ]:
```
response.candidates[0].content
```

Out[31]:
```
parts {
  text: "Computers operate through a fascinating interplay of hardware and so
ftware components.  At their core, computers rely on the central processing u
nit (CPU), which acts as the brain, executing instructions and performing cal
culations.  These instructions come from software programs, which are sets of
coded commands that tell the computer what to do.  The computer\'s memory, in
cluding RAM and storage drives, plays a crucial role in holding both data and
instructions for the CPU to access. Input and output devices, such as keyboar
ds, mice, and monitors, allow users to interact with the computer, providing
data and receiving results.  All these components work together in a coordina
ted manner, following the principles of logic and algorithms, to process info
rmation and accomplish tasks as instructed by the user. \n"
}
role: "model"
```

In [ ]:
```
result = genai.embed_content(
    model = 'models/embedding-001',
    content = response.candidates[0].content)

# 1 input > 1 vector output
print(str(result['embedding'])[:50], '... TRIMMED ...')
```

```
[0.00640031, -0.036593847, 0.011299909, 0.02542511 ... TRIMMED ...
```

Similarly, the chat history contains a list of `glm.Content` objects, which you can pass directly to the `embed_content` function:

In [ ]:
```
1 chat.history
```

Out[33]:
```
[parts {
   text: "In one sentence, explain how a computer works to a young child."
 }
 role: "user",
 parts {
   text: "Just like you use your brain to think, a computer uses its special
brain, called a CPU, to follow instructions and solve problems! \n"
 }
 role: "model",
 parts {
   text: "Okay, how about a more detailed explanation to a high schooler?"
 }
 role: "user",
 parts {
   text: "Computers operate through a fascinating interplay of hardware and s
oftware components.  At their core, computers rely on the central processing
unit (CPU), which acts as the brain, executing instructions and performing ca
lculations.  These instructions come from software programs, which are sets o
f coded commands that tell the computer what to do.  The computer\'s memory,
including RAM and storage drives, plays a crucial role in holding both data a
nd instructions for the CPU to access. Input and output devices, such as keyb
oards, mice, and monitors, allow users to interact with the computer, providi
ng data and receiving results.  All these components work together in a coord
inated manner, following the principles of logic and algorithms, to process i
nformation and accomplish tasks as instructed by the user. \n"
 }
 role: "model"]
```

In [ ]:
```
1 result = genai.embed_content(
2     model = 'models/embedding-001',
3     content = chat.history)
4
5 # 1 input > 1 vector output
6 for i,v in enumerate(result['embedding']):
7   print(str(v)[:50], '... TRIMMED...')
```

```
[-0.014632266, -0.042202696, -0.015757175, 0.01548 ... TRIMMED...
[0.0048651784, -0.06437549, -0.006556684, -0.00086 ... TRIMMED...
[-0.010055617, -0.07208932, -0.00011750793, -0.023 ... TRIMMED...
[0.00640031, -0.036593847, 0.011299909, 0.02542511 ... TRIMMED...
```

Advanced use cases

The following sections discuss advanced use cases and lower-level details of the Python SDK for the Gemini API.

Safety settings

The `safety_settings` argument lets you configure what the model blocks and allows in both prompts and responses. By default, safety settings block content with medium and/or high probability of being unsafe content across all dimensions. Learn more about Safety settings

(https://ai.google.dev/docs/safety_setting).

Enter a questionable prompt and run the model with the default safety settings, and it will not return any candidates:

In [ ]:
```
response = model.generate_content('[Questionable prompt here]')
response.candidates
```

Out[32]:
```
[content {
  parts {
    text: "I\'m sorry, but this prompt involves a sensitive topic and I\'m no
t allowed to generate responses that are potentially harmful or inappropriat
e."
  }
  role: "model"
}
finish_reason: STOP
index: 0
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
}
]
```

The `prompt_feedback` will tell you which safety filter blocked the prompt:

In [ ]:
```
1 response.prompt_feedback
```

Out[33]:
```
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
}
```

Now provide the same prompt to the model with newly configured safety settings, and you may get a response.

In [ ]:
```
1 response = model.generate_content('[Questionable prompt here]',
2                                   safety_settings={'HARASSMENT':'block_non
3 response.text
```

Also note that each candidate has its own `safety_ratings`, in case the prompt passes but the individual responses fail the safety checks.

Encode messages

The previous sections relied on the SDK to make it easy for you to send prompts to the API. This section offers a fully-typed equivalent to the previous example, so you can better understand the lower-level details regarding how the SDK encodes messages.

Underlying the Python SDK is the `google.ai.generativelanguage` (https://ai.google.dev/api/python/google/ai/generativelanguage) client library:

In [ ]:
```
1 import google.ai.generativelanguage as glm
```

The SDK attempts to convert your message to a `glm.Content` object, which contains a list of `glm.Part` objects that each contain either:

1. a `text` (https://www.tensorflow.org/text/api_docs/python/text) (string)
2. `inline_data` (`glm.Blob`), where a blob contains binary `data` and a `mime_type`.

You can also pass any of these classes as an equivalent dictionary.

Note: The only accepted mime types are some image types, `image/*`.

So, the fully-typed equivalent to the previous example is:

In [ ]:
```
model = genai.GenerativeModel('gemini-pro-vision')
response = model.generate_content(
    glm.Content(
        parts = [
            glm.Part(text="Write a short, engaging blog post based on this
            glm.Part(
                inline_data=glm.Blob(
                    mime_type='image/jpeg',
                    data=pathlib.Path('image.jpg').read_bytes()
                )
            ),
        ],
    ),
    stream=True)
```

In [ ]:
```
response.resolve()

to_markdown(response.text[:100] + "... [TRIMMED] ...")
```

Out[38]: <IPython.core.display.Markdown object>

Multi-turn conversations

While the `genai.ChatSession` class shown earlier can handle many use cases, it does make some assumptions. If your use case doesn't fit into this chat implementation it's good to remember that `genai.ChatSession` is just a wrapper around `GenerativeModel.generate_content` (https://ai.google.dev/api/python/google/generativeai/GenerativeModel#generate_content). In addition to single requests, it can handle multi-turn conversations.

The individual messages are `glm.Content` objects or compatible dictionaries, as seen in previous sections. As a dictionary, the message requires `role` and `parts` keys. The `role` in a conversation can either be the `user`, which provides the prompts, or `model`, which provides the responses.

Pass a list of `glm.Content` objects and it will be treated as multi-turn chat:

In [ ]:
```
model = genai.GenerativeModel('gemini-1.5-pro-latest')

messages = [
    {'role':'user',
     'parts': ["Briefly explain how a computer works to a young child."]}
]
response = model.generate_content(messages)

to_markdown(response.text)
```

Out[38]: <IPython.core.display.Markdown object>

To continue the conversation, add the response and another message.

Note: For multi-turn conversations, you need to send the whole conversation history with each request. The API is **stateless**.

In [ ]:
```
messages.append({'role':'model',
                 'parts':[response.text]})

messages.append({'role':'user',
                 'parts':["Okay, how about a more detailed explanation to

response = model.generate_content(messages)

to_markdown(response.text)
```

Out[39]: <IPython.core.display.Markdown object>

Generation configuration

The `generation_config` argument allows you to modify the generation parameters. Every prompt you send to the model includes parameter values that control how the model generates responses.

In [ ]:
```
model = genai.GenerativeModel('gemini-1.5-pro-latest')
response = model.generate_content(
    'Tell me a story about a magic backpack.',
    generation_config=genai.types.GenerationConfig(
        # Only one candidate for now.
        candidate_count=1,
        stop_sequences=['x'],
        max_output_tokens=20,
        temperature=1.0)
)
```

In [ ]:
```
text = response.text

if response.candidates[0].finish_reason.name == "MAX_TOKENS":
    text += '...'

to_markdown(text)
```

Out[40]: <IPython.core.display.Markdown object>

What's next

- Prompt design is the process of creating prompts that elicit the desired response from language models. Writing well structured prompts is an essential part of ensuring accurate, high quality responses from a language model. Learn about best practices for prompt writing (https://ai.google.dev/docs/prompt_best_practices).

- Gemini offers several model variations to meet the needs of different use cases, such as input types and complexity, implementations for chat or other dialog language tasks, and size constraints. Learn about the available Gemini models (https://ai.google.dev/models/gemini).
- Gemini offers options for requesting rate limit increases (https://ai.google.dev/docs/increase_quota). The rate limit for Gemini-Pro models is 60