curl \

-H 'Content-Type: application/json' \

-d '{"contents":[{"parts":[{"text":"Explain how AI works"}]}]}' \

-X POST 'https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash-latest:generateContent?key= AIzaSyCPWn048tdZrhqKuYxvRr6cu-Bjb2eAj2I’

API\_KEY = AIzaSyCPWn048tdZrhqKuYxvRr6cu-Bjb2eAj2I

# Install SDK:

%pip install -U -q "google-generativeai>=0.8.3"

import google.generativeai as genai

from IPython.display import HTML, Markdown, display

## Setup API Key:

from kaggle\_secrets import UserSecretsClient

GOOGLE\_API\_KEY = UserSecretsClient().get\_secret("GOOGLE\_API\_KEY")

genai.configure(api\_key=GOOGLE\_API\_KEY)

## First Prompt:

flash = genai.GenerativeModel('gemini-1.5-flash')

response = flash.generate\_content("Explain AI to me like I'm a kid.")

print(response.text)

## Start a Chat:

chat = flash.start\_chat(history=[])

response = chat.send\_message('Hello! My name is Prateek.')

print(response.text)

response = chat.send\_message('Can you tell something interesting about dinosaurs?')

print(response.text)

# While you have the `chat` object around, the conversation state

# persists. Confirm that by asking if it knows my name.

response = chat.send\_message('Do you remember what my name is?')

print(response.text)

# Models:

for model in genai.list\_models():

print(model.name)

# Explore generation parameters:

## Output length:

you can specify the `max\_output\_tokens` parameter when using the Gemini API. Specifying this parameter does not influence the generation of the output tokens, so the output will not become more stylistically or textually succinct, but it will stop generating tokens once the specified length is reached.

short\_model = genai.GenerativeModel(

'gemini-1.5-flash',

generation\_config=genai.GenerationConfig(max\_output\_tokens=200))

response = short\_model.generate\_content('Write a 1000 word essay on the importance of olives in modern society.')

print(response.text)

## Temperature:

Temperature controls the degree of randomness in token selection. Higher temperatures result in a higher number of candidate tokens from which the next output token is selected,

from google.api\_core import retry

high\_temp\_model = genai.GenerativeModel(

'gemini-1.5-flash',

generation\_config=genai.GenerationConfig(temperature=2.0))

# When running lots of queries, it's a good practice to use a retry policy so your code

# automatically retries when hitting Resource Exhausted (quota limit) errors.

retry\_policy = {

"retry": retry.Retry(predicate=retry.if\_transient\_error, initial=10, multiplier=1.5, timeout=300)

}

for \_ in range(5):

response = high\_temp\_model.generate\_content('Pick a random colour... (respond in a single word)',

request\_options=retry\_policy)

if response.parts:

print(response.text, '-' \* 25)

## Top-K and top-P:

Top-K is a positive integer that defines the number of most probable tokens from which to select the output token. A top-K of 1 selects a single token, performing greedy decoding.

Top-P defines the probability threshold that, once cumulatively exceeded, tokens stop being selected as candidates. A top-P of 0 is typically equivalent to greedy decoding, and a top-P of 1 typically selects every token in the model's vocabulary.

When both are supplied, the Gemini API will filter top-K tokens first, then top-P and then finally sample from the candidate tokens using the supplied temperature.

model = genai.GenerativeModel(

'gemini-1.5-flash-001',

generation\_config=genai.GenerationConfig(

# These are the default values for gemini-1.5-flash-001.

temperature=1.0,

top\_k=64,

top\_p=0.95,

))

story\_prompt = "You are a creative writer. Write a short story about a cat who goes on an adventure."

response = model.generate\_content(story\_prompt, request\_options=retry\_policy)

print(response.text)

# Prompting:

## Zero-shot

Zero-shot prompts are prompts that describe the request for the model directly.

model = genai.GenerativeModel(

'gemini-1.5-flash-001',

generation\_config=genai.GenerationConfig(

temperature=0.1,

top\_p=1,

max\_output\_tokens=5,

))

zero\_shot\_prompt = """Classify movie reviews as POSITIVE, NEUTRAL or NEGATIVE.

Review: "Her" is a disturbing study revealing the direction

humanity is headed if AI is allowed to keep evolving,

unchecked. I wish there were more movies like this masterpiece.

Sentiment: """

response = model.generate\_content(zero\_shot\_prompt, request\_options=retry\_policy)

print(response.text)

### Enum mode:

import enum

class Sentiment(enum.Enum):

POSITIVE = "positive"

NEUTRAL = "neutral"

NEGATIVE = "negative"

model = genai.GenerativeModel(

'gemini-1.5-flash-001',

generation\_config=genai.GenerationConfig(

response\_mime\_type="text/x.enum",

response\_schema=Sentiment

))

response = model.generate\_content(zero\_shot\_prompt, request\_options=retry\_policy)

print(response.text)

## One-shot and few-shot:

model = genai.GenerativeModel(

'gemini-1.5-flash-latest',

generation\_config=genai.GenerationConfig(

temperature=0.1,

top\_p=1,

max\_output\_tokens=250,

))

few\_shot\_prompt = """Parse a customer's pizza order into valid JSON:

EXAMPLE:

I want a small pizza with cheese, tomato sauce, and pepperoni.

JSON Response:

```

{

"size": "small",

"type": "normal",

"ingredients": ["cheese", "tomato sauce", "peperoni"]

}

```

EXAMPLE:

Can I get a large pizza with tomato sauce, basil and mozzarella

JSON Response:

```

{

"size": "large",

"type": "normal",

"ingredients": ["tomato sauce", "basil", "mozzarella"]

}

ORDER:

"""

customer\_order = "Give me a large with cheese & pineapple"

response = model.generate\_content([few\_shot\_prompt, customer\_order], request\_options=retry\_policy)

print(response.text)

## JSON mode:

To provide control over the schema, and to ensure that you only receive JSON (with no other text or markdown), you can use the Gemini API's [JSON mode](https://github.com/google-gemini/cookbook/blob/main/quickstarts/JSON_mode.ipynb)

import typing\_extensions as typing

class PizzaOrder(typing.TypedDict):

size: str

ingredients: list[str]

type: str

model = genai.GenerativeModel(

'gemini-1.5-flash-latest',

generation\_config=genai.GenerationConfig(

temperature=0.1,

response\_mime\_type="application/json",

response\_schema=PizzaOrder,

))

response = model.generate\_content("Can I have a large dessert pizza with apple and chocolate")

print(response.text)

## Chain of Thought:

Direct prompting on LLMs can return answers quickly and (in terms of output token usage) efficiently, but they can be prone to hallucination. The answer may "look" correct (in terms of language and syntax) but is incorrect in terms of factuality and reasoning.

Chain-of-Thought prompting is a technique where you instruct the model to output intermediate reasoning steps, and it typically gets better results, especially when combined with few-shot examples. It is worth noting that this technique doesn't completely eliminate hallucinations, and that it tends to cost more to run, due to the increased token count.

```

prompt = """When I was 4 years old, my partner was 3 times my age. Now, I

am 20 years old. How old is my partner? Return the answer directly."""

model = genai.GenerativeModel('gemini-1.5-flash-latest')

response = model.generate\_content(prompt, request\_options=retry\_policy)

print(response.text)

```

prompt = """When I was 4 years old, my partner was 3 times my age. Now,

I am 20 years old. How old is my partner? Let's think step by step."""

response = model.generate\_content(prompt, request\_options=retry\_policy)

print(response.text)

## ReAct: Reason and act

<https://github.com/google-gemini/cookbook/blob/main/examples/Search_Wikipedia_using_ReAct.ipynb>

# Come up with more examples yourself, or take a look through https://github.com/ysymyth/ReAct/

ReAct prompt directly in the Gemini API and perform the searching steps yourself. As this prompt follows a well-defined structure, there are frameworks available that wrap the prompt into easier-to-use APIs that make tool calls automatically, such as the LangChain example from the chapter.

model\_instructions = """

Solve a question answering task with interleaving Thought, Action, Observation steps. Thought can reason about the current situation,

Observation is understanding relevant information from an Action's output and Action can be one of three types:

(1) <search>entity</search>, which searches the exact entity on Wikipedia and returns the first paragraph if it exists. If not, it

will return some similar entities to search and you can try to search the information from those topics.

(2) <lookup>keyword</lookup>, which returns the next sentence containing keyword in the current context. This only does exact matches,

so keep your searches short.

(3) <finish>answer</finish>, which returns the answer and finishes the task.

"""

example1 = """Question

Musician and satirist Allie Goertz wrote a song about the "The Simpsons" character Milhouse, who Matt Groening named after who?

Thought 1

The question simplifies to "The Simpsons" character Milhouse is named after who. I only need to search Milhouse and find who it is named after.

Action 1

<search>Milhouse</search>

Observation 1

Milhouse Mussolini Van Houten is a recurring character in the Fox animated television series The Simpsons voiced by Pamela Hayden and created by Matt Groening.

Thought 2

The paragraph does not tell who Milhouse is named after, maybe I can look up "named after".

Action 2

<lookup>named after</lookup>

Observation 2

Milhouse was named after U.S. president Richard Nixon, whose middle name was Milhous.

Thought 3

Milhouse was named after U.S. president Richard Nixon, so the answer is Richard Nixon.

Action 3

<finish>Richard Nixon</finish>

"""

example2 = """Question

What is the elevation range for the area that the eastern sector of the Colorado orogeny extends into?

Thought 1

I need to search Colorado orogeny, find the area that the eastern sector of the Colorado orogeny extends into, then find the elevation range of the area.

Action 1

<search>Colorado orogeny</search>

Observation 1

The Colorado orogeny was an episode of mountain building (an orogeny) in Colorado and surrounding areas.

Thought 2

It does not mention the eastern sector. So I need to look up eastern sector.

Action 2

<lookup>eastern sector</lookup>

Observation 2

The eastern sector extends into the High Plains and is called the Central Plains orogeny.

Thought 3

The eastern sector of Colorado orogeny extends into the High Plains. So I need to search High Plains and find its elevation range.

Action 3

<search>High Plains</search>

Observation 3

High Plains refers to one of two distinct land regions

Thought 4

I need to instead search High Plains (United States).

Action 4

<search>High Plains (United States)</search>

Observation 4

The High Plains are a subregion of the Great Plains. From east to west, the High Plains rise in elevation from around 1,800 to 7,000 ft (550 to 2,130m).

Thought 5

High Plains rise in elevation from around 1,800 to 7,000 ft, so the answer is 1,800 to 7,000 ft.

Action 5

<finish>1,800 to 7,000 ft</finish>

"""

To capture a single step at a time, while ignoring any hallucinated Observation steps, you will use `stop\_sequences` to end the generation process.

question = """Question

Who was the youngest author listed on the transformers NLP paper?

"""

model = genai.GenerativeModel('gemini-1.5-flash-latest')

react\_chat = model.start\_chat()

# You will perform the Action, so generate up to, but not including, the Observation.

config = genai.GenerationConfig(stop\_sequences=["\nObservation"])

resp = react\_chat.send\_message(

[model\_instructions, example1, example2, question],

generation\_config=config,

request\_options=retry\_policy)

print(resp.text)

# Code prompting:

## Generating code:

model = genai.GenerativeModel(

'gemini-1.5-flash-latest',

generation\_config=genai.GenerationConfig(

temperature=1,

top\_p=1,

max\_output\_tokens=1024,

))

# Gemini 1.5 models are very chatty, so it helps to specify they stick to the code.

code\_prompt = """

Write a Python function to calculate the factorial of a number. No explanation, provide only the code.

"""

response = model.generate\_content(code\_prompt, request\_options=retry\_policy)

Markdown(response.text)

## Code execution:

model = genai.GenerativeModel(

'gemini-1.5-flash-latest',

tools='code\_execution',)

code\_exec\_prompt = """

Calculate the sum of the first 14 prime numbers. Only consider the odd primes, and make sure you count them all.

"""

response = model.generate\_content(code\_exec\_prompt, request\_options=retry\_policy)

Markdown(response.text)

## Explaining Code:

file\_contents = !curl https://raw.githubusercontent.com/magicmonty/bash-git-prompt/refs/heads/master/gitprompt.sh

explain\_prompt = f"""

Please explain what this file does at a very high level. What is it, and why would I use it?

```

{file\_contents}

```

"""

model = genai.GenerativeModel('gemini-1.5-flash-latest')

response = model.generate\_content(explain\_prompt, request\_options=retry\_policy)

Markdown(response.text)

**Learn more**

To learn more about prompting in depth:

* Check out the whitepaper issued with today's content,
* Try out the apps listed at the top of this notebook ([TextFX](https://textfx.withgoogle.com/" \t "_blank), [SQL Talk](https://sql-talk-r5gdynozbq-uc.a.run.app/) and [NotebookLM](https://notebooklm.google/" \t "_blank)),
* Read the [Introduction to Prompting](https://ai.google.dev/gemini-api/docs/prompting-intro) from the Gemini API docs,
* Explore the Gemini API's [prompt gallery](https://ai.google.dev/gemini-api/prompts) and try them out in AI Studio,
* Check out the Gemini API cookbook for [inspirational examples](https://github.com/google-gemini/cookbook/blob/main/examples/) and [educational quickstarts](https://github.com/google-gemini/cookbook/blob/main/quickstarts/).