

# ITCS 4101: Introduction to NLP

---

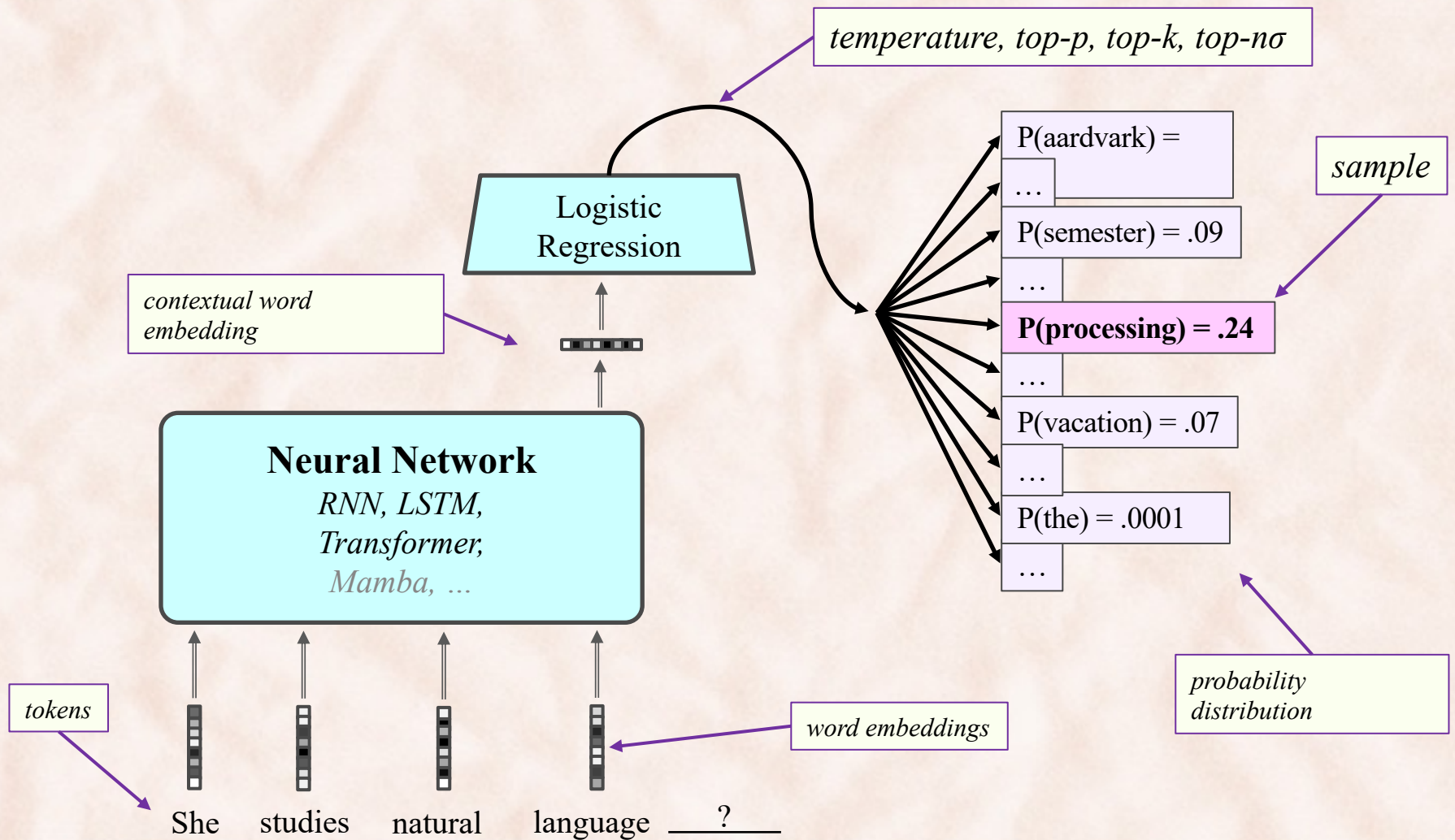
Coding with Large Language Models using APIs:  
GPT, Gemini, Qwen

Razvan C. Bunescu

Department of Computer Science @ CCI

[rbunescu@charlotte.edu](mailto:rbunescu@charlotte.edu)

# Large Language Models (LLMs)



# Application Development Using LLM APIs

---

We will discuss 3 main options:

1. OpenAI's **GPT** models.
  - Use as a service, pay per token.
  - Open-source versions [gpt-oss](#).
2. Google's **Gemini** models.
  - Free tier, with lower rate limits.
  - Paid tier, with higher rate limits.
  - Need personal Gmail account.
3. Open source **Qwen** models:
  - FP8 quantized [Qwen3.5-35B](#).

# Two Ways of Using GPT and Gemini Models

---

## 1. Through the **browser app**:

- ChatGPT at <https://chatgpt.com>
- Gemini at <https://gemini.google.com/app>

## 2. As a service, through an **API**:

- **Directly**, in the code:
  - GPT through the [Response API](#).
  - Gemini through the [Developer API](#).
- **Indirectly**, in the browser:
  - GPT Playground at <https://platform.openai.com/chat>
  - Google AI Studio at [https://aistudio.google.com/prompts/new\\_chat](https://aistudio.google.com/prompts/new_chat)

# OpenAI GPT

---

# Using OpenAI GPT models

---

- GPT = Generative Pre-trained Transformer.
- **Pre-trained** to “understand” natural language and code:
  - Using a *language modeling* (LM) objective.
- **Fine-tuned** to provide text outputs (answers) in response to their inputs (questions or **prompts**).
  - **Instruction fine-tuning**.
  - **Alignment with human preference** (RLHF with PPO, DPO, ...).
- “Programming” with GPT, Gemini, Llama, and other LLMs:
  - Design a “prompt”, usually by providing **instructions** and/or some examples of how to successfully complete a task:
    - *zero-shot, few-shot in-context learning, CoT explanations.*

# GPT: Setting up the OpenAI API account

---

- Need to have an OpenAI account:
  - Go to <https://platform.openai.com>, Log in / Signup.
    - “Continue with Google”, use your UNCC email.
      - \$5 should be enough for the work in this class, see [pricing](#).
      - Go to [billing overview](#), *Set payment* → input credit card, or *Add to credit balance*, input \$5.
- Create a secret API key and store it in a **.env** file:
  - Go to [API keys](#) and “+ create new secret key”.
  - Copy the key and store it in a text file named **.env** as follows
    - **OPENAI\_API\_KEY=...**
    - Make sure you save the key, it will not be shown again.
  - Place or copy the **.env** file in the folder you edit and run the notebook.
  - Do not put the secret key in your code!

# Required Python Modules

---

- Install the **openai** module and **python-dotenv** module:
  - pip install openai
  - pip install python-dotenv

```
import os
from openai import OpenAI

from dotenv import load_dotenv, find_dotenv

# Read the local .env file, containing the Open AI secret key.
_ = load_dotenv(find_dotenv())

client = OpenAI(api_key = os.environ['OPENAI_API_KEY'])
```

- Alternatively, use Google Colab instead of JupyterLab:
  - Has modules already installed.
  - But ensure to use Colab’s built-in “Secrets” feature to store the keys.

# Setting up and reading secret keys in Colab

## 2. Accessing Colab Secrets (Recommended for sensitive data):

For sensitive information like API keys, it is recommended to use Colab's built-in "Secrets" feature.

### • Set up the Secret:

- In your Colab notebook, click the "key" icon on the left sidebar to open the Secrets manager.
- Add a new secret, providing a name (e.g., `MY_API_KEY`) and its corresponding value.
- Ensure the "Notebook access" toggle is enabled for your current notebook.

### • Read the Secret in your Notebook:

```
Python
from google.colab import userdata
import os

# Retrieve the secret value using its name
api_key = userdata.get('MY_API_KEY')

# Optionally, set it as an environment variable for compatibility with
os.environ['MY_API_KEY'] = api_key

print(f"API Key (first few chars): {api_key[:5]}...") # Print a part
```

## 3. Reading from a `.env` file (if you've uploaded one):

If you've uploaded a `.env` file to your Colab environment, you can use the `python-dotenv` library to load it:

```
Python
!pip install python-dotenv

import dotenv
dotenv.load_dotenv('./.env')

import os
value = os.environ.get('YOUR_VARIABLE_NAME_IN_DOTENV')
print(f"The value from .env is: {value}")
```

ExamplesGPT.ipynb ☆ ☁

File Edit View Insert Runtime Tools Help

🔍 Commands | + Code + Text | ▶ Run all ▾

☰ Secrets 📄 ✕

🔍 Configure your code by storing environment variables, file paths, or keys. Values stored here are private, visible only to you and the notebooks that you select.

⏪ ⏩ Secret name cannot contain spaces.

🔑 Notebook access

Name	Value	Actions
OPENAI_A	sk-proj-LV	<input checked="" type="checkbox"/> 🗑️ 📄

+ Add new secret

Gemini API keys ▾

Access your secret keys in Python via:

```
from google.colab import userdata
userdata.get('secretName')
```

# Using the Response API

---

- Take a list of messages as input and return a model-generated message as output.
  - Designed to make multi-turn conversations easy, it's just as useful for single-turn tasks without any conversation.

```
response1 = client.responses.create(  
    model = "gpt-5-nano",  
    reasoning = {"effort": "low"},  
    input = [  
        {  
            "role": "developer",  
            "content": "You are a helpful music historian."  
        },  
        {  
            "role": "user",  
            "content": "Who composed The Four Seasons?"  
        }  
    ]  
)  
  
print(response1.output_text)
```

<https://platform.openai.com/docs/guides/text>

# Response API: `openai.responses.create`

---

- 3 major roles in the **input** parameter:
  - **Developer**: Optional message, that indicates the LLM persona.
    - Also called a *steering prompt*, sets up the system behavior.
  - **User**: Provides questions, requests, or comments to the assistant.
  - **Assistant**: Previous responses from the LM assistant, or example of desired LM response.
    - **Need to provide the conversation so far, every time** we want to continue with a new user questions.
- Roles interact in a chain of command, with authority levels:

1. **Platform**: Model Spec "platform" sections and system messages
2. **Developer**: Model Spec "developer" sections and developer messages
3. **User**: Model Spec "user" sections and user messages
4. **Guideline**: Model Spec "guideline" sections
5. *No Authority*: assistant and tool messages; quoted/untrusted text and multimodal data in other messages

# Response API: `openai.responses.create`

---

- Other useful parameters:
  - **model**: `gpt-5` or `gpt-5-min` or `gpt-5-nano` or `gpt-5.2-pro` or `gpt-5.2-codex` or ...
  - **max\_output\_tokens**: maximum # of tokens to generate.
  - **reasoning**: effort level and configuration for reasoning models, default is 'medium'.
  - **tool\_choice**: Controls which (if any) tool is called by the model.
  - **temperature**: *used to default to 1, and set it to **0 for greedy decoding**.*
    - Eliminated in GPT-5!
  - **top\_p**: defaults to 1, use 0.1 if you want the LLM to sample tokens only from the top 10% of probability mass, i.e. **nucleus sampling**.
  - **presence\_penalty**, **frequency\_penalty**, **logit\_bias**: penalize or favor repetitions, or certain tokens (later in this course).

<https://platform.openai.com/docs/api-reference/responses>

# Examples with Open AI GPT

---

- Shown in the Jupyter notebook.

# Using the API interactively in GPT Playground

The screenshot displays the GPT Playground interface. At the top, there's a navigation bar with a back arrow, the title "4101 example 1", a dropdown menu set to "v1 · default", and action buttons for "Compare", "Optimize", "Evaluate", and "Update".

The left sidebar contains configuration options: "Model" is set to "gpt-5-nano" with a dropdown arrow and a settings icon; "Variables" and "Tools" each have a "+ Add" button; and "Developer message" has a settings icon.

The main content area is split into two columns. The left column shows a "Developer message" box with the text: "You are an expert software engineer who routinely uses AI to increase productivity." Below this is a "Prompt messages" section with a "User" message: "Enter task specifics. Use {{template variables}} for dynamic inputs". At the bottom of this section is a "+ Add message" button.

The right column shows the "Assistant" response. It starts with "Here are 10 essential skills to land a software engineering role that involves AI:" followed by a numbered list:

1. Core CS and software engineering fundamentals
  - Data structures, algorithms, complexity, clean code, testing, debugging, and version control (Git).
2. Math for AI and ML basics
  - Linear algebra, calculus, probability, statistics; ML concepts (supervised/unsupervised learning, evaluation metrics, bias-variance).
3. Proficiency with ML frameworks and tooling
  - Python + libraries (NumPy, pandas, scikit-learn) and deep learning frameworks (PyTorch or TensorFlow); experience with experiment tracking (MLflow, Weights & Biases).
4. End-to-end ML pipelines and MLOps
  - Data acquisition/labeling, preprocessing, feature engineering, model training, evaluation, deployment, monitoring. (22.9s ↑ 58t ↓ 2,394t), reproducibility.

At the bottom of the right column is a "Chat with your prompt..." input field with a settings icon, an "Auto-clear" button, and a send button.

# Using the API interactively in GPT Playground

- The Playground facilitates quick stress testing of prompts and parameters, before deploying in code.
- Once the prompt and parameters are ready, you can see the Python or node.js code for the conversation. Copy & Paste into your code.

```
POST /v1/responses python 
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.responses.create(
5     model="gpt-5-nano",
6     input=[
7         {
8             "role": "developer",
9             "content": [
10                {
11                    "type": "input_text",
12                    "text": "You are an expert software engineer who routinely uses /
13                }
14            ]
15        },
16        {
17            "role": "user",
18            "content": [
19                {
20                    "type": "input_text",
21                    "text": "As a CS student, what are the top 10 skills I should lea
22                }
23            ]
24        },
25        {
26            "type": "reasoning",
27            "id": "rs_68cc2fbd0f08197912a6f5c842d36340e0d485508b3601a",
28            "summary": [],
29            "encrypted_content": "gAAAAABozC_SSCaoo8c0-_y2wTSGx058A9wxya05wMgLL_L
30        },
31        {
32            "id": "msg_68cc2fcf13508197afd74d2f126b3e6a0e0d485508b3601a",
33            "role": "assistant",
34            "content": [
35                {
36                    "type": "output_text",
37                    "text": "Here are 10 essential skills to land a software enginee
38                }
39            ]
40        }
41    ],
42    text={
43        "format": {
44            "type": "text"
45        },
46        "verbosity": "low"
47    },
48    reasoning={
49        "effort": "medium"
50    },
51    tools=[],
52    store=True,
53    include=[
54        "reasoning.encrypted_content",
55        "web_search_call.action.sources"
56    ]
57 )
```

# Google Gemini

---

# Gemini: Setting up the API account

- Need to have a personal Google account:

- UNCC account will not work (ask OIT ...)

We are sorry, but **you do not have access to Google Developers**. Some reasons why you may not have access:

- Your account is managed by an organization that has this service turned off for its users.
- Your account may be temporarily disabled by your [organization's admin](#).

- Go to <https://ai.google.dev/>, Sign in.

- Continue with **personal account**.

- See [pricing](#) for available models and pricing and [rate limits](#).

## Gemini 2.5 Flash

*gemini-2.5-flash*

Try it in Google AI Studio

Our first hybrid reasoning model which supports a 1M token context window and has thinking budgets.

	Standard	Batch
	Free Tier	Paid Tier, per 1M tokens in USD
Input price	Free of charge	\$0.30 (text / image / video) \$1.00 (audio)
Output price (including thinking tokens)	Free of charge	\$2.50
Context caching price	Not available	\$0.03 (text / image / video) \$0.1 (audio) \$1.00 / 1,000,000 tokens per hour (storage price)
Grounding with Google Search	Free of charge, up to 500 RPD (limit shared with Flash-Lite RPD)	1,500 RPD (free, limit shared with Flash-Lite RPD), then \$35 / 1,000 grounded prompts
Grounding with Google Maps	500 RPD	1,500 RPD (free), then \$25 / 1,000 grounded prompts

## Gemini 2.5 Pro

*gemini-2.5-pro*

Try it in Google AI Studio

Our state-of-the-art multipurpose model, which excels at coding and complex reasoning tasks.




	Standard	Batch
	Free Tier	Paid Tier, per 1M tokens in USD
Input price	Free of charge	\$1.25, prompts <= 200k tokens \$2.50, prompts > 200k tokens
Output price (including thinking tokens)	Free of charge	\$10.00, prompts <= 200k tokens \$15.00, prompts > 200k
Context caching price	Not available	\$0.125, prompts <= 200k tokens \$0.25, prompts > 200k \$4.50 / 1,000,000 tokens per hour (storage price)
Grounding with Google Search	Not available	1,500 RPD (free), then \$35 / 1,000 grounded prompts
Grounding with Google Maps	Not available	10,000 RPD (free), then \$25 / 1,000 grounded prompts

# Gemini: Setting up the API account

---

- First time API users get \$300 free credit for Google Cloud:
  - Including the Gemini API.
  - To be used within 3 months.

Beginning today, you have \$300 USD in credit which you can use to:

-  Evaluate Google Cloud **risk-free\***
-  Explore a wide range of Google Cloud products and services – from [Compute Engine](#) and [BigQuery](#) to [industry-leading AI](#).
-  Easily check your credit usage by visiting the [billing section](#) of your Google Cloud console

# Gemini: One-time Setup of API Key

---

- Create and store a secret API key:
  - Get an API key from [Google AI Studio](#).
    - Click **Create API Key** at <https://ai.google.dev/gemini-api/docs/api-key>
  - Copy the key and store it in a text file named **.env** as follows
    - **GEMINI\_API\_KEY=...**
- Place or copy the **.env** file in the folder you edit and run the notebook.
  - Other solutions exist, but this is what we will do in this course.
  - Do not put the secret key in your code!

# Required Python Modules

---

- Install the **google-genai** module:
  - pip install -U google-genai (use pip3)
    - Make sure you have latest version of pip3 and setuptools:
      - pip3 install --upgrade pip
      - python3 -m pip install --upgrade setuptools
- Install the **python-dotenv** module, using one of:
  - pip install python-dotenv
- Alternatively, use [Colab](#) instead of Jupyter Lab / Notebook:
  - Has modules already installed.
  - But ensure to use Colab's built-in "Secrets" feature to store the keys.

# Setting up and reading secret keys in Colab

## 2. Accessing Colab Secrets (Recommended for sensitive data):

For sensitive information like API keys, it is recommended to use Colab's built-in "Secrets" feature.

### • Set up the Secret:

- In your Colab notebook, click the "key" icon on the left sidebar to open the Secrets manager.
- Add a new secret, providing a name (e.g., `MY_API_KEY`) and its corresponding value.
- Ensure the "Notebook access" toggle is enabled for your current notebook.

### • Read the Secret in your Notebook:

```
Python
from google.colab import userdata
import os

# Retrieve the secret value using its name
api_key = userdata.get('MY_API_KEY')

# Optionally, set it as an environment variable for compatibility with
os.environ['MY_API_KEY'] = api_key

print(f"API Key (first few chars): {api_key[:5]}...") # Print a parti
```

## 3. Reading from a `.env` file (if you've uploaded one):

If you've uploaded a `.env` file to your Colab environment, you can use the `python-dotenv` library to load it:

```
Python
!pip install python-dotenv

import dotenv
dotenv.load_dotenv('./.env')

import os
value = os.environ.get('YOUR_VARIABLE_NAME_IN_DOTENV')
print(f"The value from .env is: {value}")
```

ExamplesGemini.ipynb ☆ ☁

File Edit View Insert Runtime Tools Help

🔍 Commands | + Code | + Text | ▶ Run all ▾

### Secrets

Configure your code by storing environment variables, file paths, or keys. Values stored here are private, visible only to you and the notebooks that you select.

<>

Secret name cannot contain spaces.

Notebook access	Name	Value	Actions
<input type="checkbox"/>	OPENAI_A	.....	👁️ 📄 🗑️
<input type="checkbox"/>	GEMINI_AI	.....	👁️ 📄 🗑️

+ Add new secret

Gemini API keys ▾

Access your secret keys in Python via:

```
from google.colab import userdata
userdata.get('secretName')
```

# Gemini Client Setup and Query

---

- Create the Client object:

```
import os
from google import genai
from dotenv import load_dotenv, find_dotenv

# Read the local .env file, containing the Gemini secret API key.
_ = load_dotenv(find_dotenv())

client = genai.Client(api_key = os.environ["GEMINI_API_KEY"])
```

- Send a query, using default parameters:

```
query = "Justin sits next to Razvan. One of them is happy and one of them is grumpy. " \
        "The person sitting next to Justin is grumpy. Who is happy?"

response = client.models.generate_content(
    model = "gemini-2.5-flash",
    contents = query
)
print(response.text)
```

# Gemini: Multiple Turn Conversations

---

- Use the **google.genai.chats.Chat** class to manage turns in the conversation:

```
chat = client.chats.create(model = "gemini-2.5-flash")

response = chat.send_message("Who received the Nobel prize in medicine for cancer immunotherapy?")
print(response.text)

response = chat.send_message("Where did they do their research?")
print(response.text)

for message in chat.get_history():
    print(f'role - {message.role}', end = ": ")
    print(message.parts[0].text)
```

★ **Note:** Chat functionality is only implemented as part of the SDKs. Behind the scenes, it still uses the [generateContent](#) API. For multi-turn conversations, the full conversation history is sent to the model with each follow-up turn.

Hard to find API documentation on how to engage in **multi-turn multimodal conversations** ...

# When API Documentation Fails, Turn to Google's AI Overview

gemini api generate\_content with multiple content fields

AI Mode **All** Videos Images Short videos Forums Shopping More ▾ Tools

## ◆ AI Overview

The Gemini API's `generateContent` method allows for the inclusion of multiple content fields within a single request, particularly when dealing with multimodal inputs or multi-turn conversations.

### Multimodal Inputs:

Gemini models are designed to handle various modalities like text, images, and audio. To include multiple content types in a single `generateContent` call, you structure the `contents` field of your request to contain a list of `Part` objects, each representing a different modality.

For example, to send both text and an image:

Code

```
{
  "contents": [
    {
      "role": "user",
      "parts": [
        {
          "text": "Describe this image:"
        },
        {
          "inlineData": {
            "mimeType": "image/jpeg",
            "data": "base64_encoded_image_data"
          }
        }
      ]
    }
  ]
}
```

### Multi-Turn Conversations (Chat):

When working with chat-like interactions, you can provide the conversation history by including multiple `Content` objects within the `contents` list. Each `Content` object represents a turn in the conversation, specifying the `role` (e.g., "user" or "model") and the `parts` of that turn.

Code

```
{
  "contents": [
    {
      "role": "user",
      "parts": [
        {
          "text": "What is the capital of France?"
        }
      ]
    },
    {
      "role": "model",
      "parts": [
        {
          "text": "The capital of France is Paris."
        }
      ]
    },
    {
      "role": "user",
      "parts": [
        {
          "text": "Tell me more about its history."
        }
      ]
    }
  ]
}
```

# Turn to AI Mode



google gemini api how to set the content parts in Python



AI Mode

All

Videos

Short videos

Images

Forums

Shopping

More ▾

Tools ▾

## google gemini api how to set the content parts in Python

To set content parts for a multimodal prompt with the Google Gemini API in Python, **create a list of items**. Pass this list to the `contents` parameter of the `generate_content` method. This list can contain text strings, uploaded file objects, or inline data objects.

### Supported content types

The parts in the `contents` list can include various data types for different modalities:

- **Text:** Simple Python strings.
- **Inline data:** Base64-encoded bytes with a MIME type. This is suitable for smaller files (less than 20MB).
- **File data:** Use the `client.files.upload()` method to upload files and then reference the returned file object in your prompt. This is recommended for larger files.
- **YouTube URLs:** For video understanding, you can pass YouTube URLs directly.

AI responses may include mistakes. [Learn more](#)



↳ Show me how to send inline data with a MIME type

↳ What are the MIME types Gemini API supports?

↳ Tell me more about the File API

### Example: Multimodal chat conversation

For multi-turn chats, the `contents` list contains a series of `Content` objects, with alternating roles for "user" and "model".

### python

```
import google.generativeai as genai

model = genai.GenerativeModel("gemini-2.5-flash")

# Create a chat session with initial history
chat_session = model.start_chat(history=[
    {
        "role": "user",
        "parts": ["What are the ingredients in this image?"]
    },
    {
        "role": "model",
        "parts": ["There is flour, eggs, and chocolate chips."]
    },
])

# Send a new message that includes a file
# First, upload the image file
myfile = genai.Client().files.upload(path="path/to/new_image.jpg")

# Then, send a new message with multiple content parts
response = chat_session.send_message(
    contents=[
        myfile,
        "Is there a different kind of flour in this picture?"
    ]
)

print(response.text)
```

# Text Embeddings

- We can also embed a piece of text using [Gemini embeddings](#):
  - Generate embeddings for *words, phrases, sentences, and code*.

```
from google import genai
from google.genai import types
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

client = genai.Client()

texts = [
    "What is the meaning of life?",
    "What is the purpose of existence?",
    "How do I bake a cake?"
]

result = [
    np.array(e.values) for e in client.models.embed_content(
        model="gemini-embedding-001",
        contents=texts,
        config=types.EmbedContentConfig(task_type="SEMANTIC_SIMILARITY")).embeddings
]

# Calculate cosine similarity. Higher scores = greater semantic similarity.

embeddings_matrix = np.array(result)
similarity_matrix = cosine_similarity(embeddings_matrix)

for i, text1 in enumerate(texts):
    for j in range(i + 1, len(texts)):
        text2 = texts[j]
        similarity = similarity_matrix[i, j]
        print(f"Similarity between '{text1}' and '{text2}': {similarity:.4f}")
```

# Text Embeddings

- We can also embed a piece of text using [Gemini embeddings](#):
  - Embeddings are pretrained for various tasks:

Task type	Description	Examples
SEMANTIC_SIMILARITY	Embeddings optimized to assess text similarity.	Recommendation systems, duplicate detection
CLASSIFICATION	Embeddings optimized to classify texts according to preset labels.	Sentiment analysis, spam detection
CLUSTERING	Embeddings optimized to cluster texts based on their similarities.	Document organization, market research, anomaly detection
RETRIEVAL_DOCUMENT	Embeddings optimized for document search.	Indexing articles, books, or web pages for search.
RETRIEVAL_QUERY	Embeddings optimized for general search queries. Use <b>RETRIEVAL_QUERY</b> for queries; <b>RETRIEVAL_DOCUMENT</b> for documents to be retrieved.	Custom search
CODE_RETRIEVAL_QUERY	Embeddings optimized for retrieval of code blocks based on natural language queries. Use <b>CODE_RETRIEVAL_QUERY</b> for queries; <b>RETRIEVAL_DOCUMENT</b> for code blocks to be retrieved.	Code suggestions and search
QUESTION_ANSWERING	Embeddings for questions in a question-answering system, optimized for finding documents that answer the question. Use <b>QUESTION_ANSWERING</b> for questions; <b>RETRIEVAL_DOCUMENT</b> for documents to be retrieved.	Chatbox
FACT_VERIFICATION	Embeddings for statements that need to be verified, optimized for retrieving documents that contain evidence supporting or refuting the statement. Use <b>FACT_VERIFICATION</b> for the target text; <b>RETRIEVAL_DOCUMENT</b> for documents to be retrieved	Automated fact-checking systems

# Examples with Google Gemini

---

- Shown in the Jupyter notebook.
- More examples in the [Gemini Developer API](#) documentation:
  - [Text generation](#) examples.
  - [Image generation](#) examples.
  - [Image understanding](#) example.
- Parameters such as `temperature`, `max_output_tokens`, ... can be set using [`google.genai.types.GenerateContentConfig\(\)`](#).
- More documentation available at:
  - [Google Gen AI SDK](#).
  - [Vertex AI](#).

# Qwen

---

# Using Qwen 3.5-35B FP8 Quantized

---

- **OpenAI.base\_url:**
  - An attribute of the OpenAI class.
- **Model name:**
  - Specifies which version of Qwen is being utilized.
  - You must be on [eduroam](#) to access the model directly. Off campus, you need to connect through the educational cluster using VPN.

```
import httpx
from openai import OpenAI

# Set the Qwen API base URL.
BASE_URL = "https://cci-llm.charlotte.edu/api/v1"

# Initialize client with SSL verification disabled
client = OpenAI(base_url = BASE_URL,
                http_client = httpx.Client(verify = False),
                api_key = '3jdhdi4xkf-45')

model_name = "Qwen3.5-35B-A3B-FP8"
```

- Send messages using the original **chat completion API** from OpenAI.

# Chat Completion API:

## *openai.chat.completions.create()*

---

- Initial API from OpenAI, still has backward support for it.

```
from openai import OpenAI
client = OpenAI(api_key = os.environ['OPENAI_API_KEY'])
```

<https://platform.openai.com/docs/api-reference/chat>

- Most LLMs support it, including Qwen, Llama, and Gemini.
  - Take a list of messages as input and return a model-generated message as output.
  - Designed to make multi-turn conversations easy, it's just as useful for single-turn tasks without any conversation.

model\_name

```
response = client.chat.completions.create(
    model = 'gpt-4o',
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Who composed The Four Seasons?"},
        {"role": "assistant", "content": "Antonio Vivaldi composed The Four Seasons."},
        {"role": "user", "content": "For whom were most of his compositions written?"}
    ]
)

print(response.choices[0].message.content)
```

# Chat Completion API:

## *openai.chat.completions.create()*

---

- 3 major roles in the **messages** parameter:
  - **System**: Optional first message, that indicates the LM persona.
    - Also called a *steering prompt*, sets up the system behavior.
    - Default is “You are a helpful assistant”.
  - **User**: Provides questions, requests, or comments to the assistant.
  - **Assistant**: Previous responses from the LM assistant, or example of desired LM response.
    - **Need to provide the conversation so far every time** we want to continue with a new user questions.
- Typical input (RE) is **system? user (assistant user)\***

# Chat Completion API:

## *openai.chat.completions.create()*

---

- Other useful parameters:
  - **model**: `gpt-5` or `gpt-5-min` or `gpt-5-nano` or ...
  - **temperature**: defaults to 1, but set it to **0 for greedy decoding**.
  - **top\_p**: defaults to 1, use 0.1 if you want the LM to sample tokens only from the top 10% of probability mass, i.e. **nucleus sampling**.
  - **n** : defaults to 1, indicates # completions (alternatives) to generate.
  - **max\_tokens**: defaults to  $\infty$ , maximum # of tokens to generate.
  - `presence_penalty`, `frequency_penalty`, `logit_bias`: penalize or favor repetitions, or certain tokens (later in this course).

# Recommended Parameters for Qwen

---

We recommend using the following set of sampling parameters for generation

- **Thinking mode for general tasks:** `temperature=1.0, top_p=0.95, top_k=20, min_p=0.0, presence_penalty=1.5, repetition_penalty=1.0`
- **Thinking mode for precise coding tasks (e.g. WebDev):** `temperature=0.6, top_p=0.95, top_k=20, min_p=0.0, presence_penalty=0.0, repetition_penalty=1.0`
- **Instruct (or non-thinking) mode for general tasks:** `temperature=0.7, top_p=0.8, top_k=20, min_p=0.0, presence_penalty=1.5, repetition_penalty=1.0`
- **Instruct (or non-thinking) mode for reasoning tasks:** `temperature=1.0, top_p=0.95, top_k=20, min_p=0.0, presence_penalty=1.5, repetition_penalty=1.0`

Please note that the support for sampling parameters varies according to inference frameworks.

<https://huggingface.co/Qwen/Qwen3.5-35B-A3B#using-qwen35-via-the-chat-completions-api>

# Using Qwen through the ccAPI

---

```
# Define the conversation
```

```
Go to line number...oy sits next to Razvan. One of them is happy and one of them is grumpy. " \
    "The person sitting next to Tonmoy is grumpy. Who is happy?"

conversation = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": query}
]
```

```
# Send a chat completion request
```

```
response = client.chat.completions.create(
    model = model_name,
    messages = conversation,
    max_tokens = 500,
    temperature = 0
)

reply = response.choices[0].message.content
print(f"Text response: {reply}")
```

# Using Qwen through the ccAPI

---

## Python & JSON Comprehension

```
question = 'Consider the following monologue from the movie Stalker by Andrei Tarkovsky: ' \
    '"Let them be helpless like children, because weakness is a great thing, and strength is nothing. ' \
    'When a man is just born, he is weak and flexible. When he dies, he is hard and insensitive. ' \
    'When a tree is growing, it\'s tender and pliant. But when it\'s dry and hard, it dies. ' \
    'Hardness and strength are death\'s companions. Pliancy and weakness are expressions of the ' \
    'freshness of being. Because what has hardened will never win.'" ' \
    'Where else was a similar idea expressed? Provide quotes. Format your answer as a Python dictionary ' \
    'mapping the author or source name to the actual passage expressing a similar idea.'

conversation = [{"role": "system", "content": "You are a helpful librarian."},
               {"role": "user",
                "content": question}]

response = client.chat.completions.create(
    model = model_name,
    messages = conversation,
    temperature = 0
)

print(response.choices[0].message.content)
```

# More Examples with Qwen 3.5

---

- Shown in the [Jupyter Lab notebook](#).