Search projects

Help Sponsors Log in Register

openai 1.59.6

✓ Latest version

```
pip install openai
```

Released: Jan 10, 2025

The official Python library for the openai API

Navigation

Project description

Release history

Download files

Verified details ✓

These details have been verified by PyPI

Maintainers

atty-openai

borispower

Project description

OpenAI Python API library

pypi v1.59.2

The OpenAI Python library provides convenient access to the OpenAI REST API from any Python 3.8+ application. The library includes type definitions for all request params and response fields, and offers both synchronous and asynchronous clients powered by httpx.

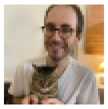
It is generated from our OpenAPI specification with Stainless.

Documentation

The REST API documentation can be found on platform.openai.com. The full API of this library can be found in api.md.

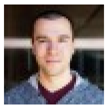
Installation

christinakim


ddeville

dschnurr-openai

emorikawa-openai


gdb

hallacy-openai

hponde_oai

jhallard

kennyhsu

michelle-openai

mikaell-openai

peterz-openai

rachel-openai

> *[!IMPORTANT] The SDK was rewritten in v1, which was released November 6th 2023. See the v1 migration guide, which includes scripts to automatically update your code.*

```
# install from PyPI
pip install openai
```

Usage

The full API of this library can be found in api.md.

```
import os
from openai import OpenAI

client = OpenAI(
    api_key=os.environ.get("OPENAI_API_KEY"),  # This is
)

chat_completion = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Say this is a test",
        }
    ],
    model="gpt-4o",
)
```

While you can provide an `api_key` keyword argument, we recommend using python-dotenv to add `OPENAI_API_KEY="My API Key"` to your `.env` file so that your API Key is not stored in source control.

Vision

With a hosted image:

tomerkOpenAI

Unverified details

*These details have **not** been verified by PyPI*

Project links

- Homepage
- Repository

Meta

- **License:** Apache Software License
- **Author:** OpenAI
- **Requires:** Python >=3.8

Classifiers

Intended Audience

- Developers

License

- OSI Approved :: Apache Software License

Operating System

- MacOS
- Microsoft :: Windows
- OS Independent
- POSIX
- POSIX :: Linux

Programming Language

- Python :: 3.8
- Python :: 3.9
- Python :: 3.10

```
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": prompt},
                {
                    "type": "image_url",
                    "image_url": {"url": f"{img_url}"},
                },
            ],
        }
    ],
)
```

With the image as a base64 encoded string:

```
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": prompt},
                {
                    "type": "image_url",
                    "image_url": {"url": f"data:{img_type
                },
            ],
        }
    ],
)
```

Polling Helpers

When interacting with the API some actions such as starting a Run and adding files to vector stores are asynchronous and take time to complete. The SDK includes helper functions which will poll the status until it reaches a terminal state and then return the resulting object. If

- Python :: 3.11
- Python :: 3.12

Topic

- Software Development :: Libraries :: Python Modules

Typing

- Typed

JetBrains is a Contributing sponsor of the Python Software Foundation.

PSF Sponsor · Served ethically

an API method results in an action that could benefit from polling there will be a corresponding version of the method ending in '_and_poll'.

For instance to create a Run and poll until it reaches a terminal state you can run:

```
run = client.beta.threads.runs.create_and_poll(
    thread_id=thread.id,
    assistant_id=assistant.id,
)
```

More information on the lifecycle of a Run can be found in the Run Lifecycle Documentation

Bulk Upload Helpers

When creating and interacting with vector stores, you can use polling helpers to monitor the status of operations. For convenience, we also provide a bulk upload helper to allow you to simultaneously upload several files at once.

```
sample_files = [Path("sample-paper.pdf"), ...]

batch = await client.vector_stores.file_batches.upload_ar
    store.id,
    files=sample_files,
)
```

Streaming Helpers

The SDK also includes helpers to process streams and handle incoming events.

```
with client.beta.threads.runs.stream(
    thread_id=thread.id,
    assistant_id=assistant.id,
    instructions="Please address the user as Jane Doe. Th
) as stream:
    for event in stream:
        # Print the text from text delta events
        if event.type == "thread.message.delta" and event
            print(event.data.delta.content[0].text)
```

More information on streaming helpers can be found in the dedicated documentation: helpers.md

Async usage

Simply import `AsyncOpenAI` instead of `OpenAI` and use `await` with each API call:

```
import os
import asyncio
from openai import AsyncOpenAI

client = AsyncOpenAI(
    api_key=os.environ.get("OPENAI_API_KEY"),  # This is
)


async def main() -> None:
    chat_completion = await client.chat.completions.creat
        messages=[
            {
                "role": "user",
                "content": "Say this is a test",
            }
        ],
        model="gpt-4o",
    )


asyncio.run(main())
```

Functionality between the synchronous and asynchronous clients is otherwise identical.

Streaming responses

We provide support for streaming responses using Server Side Events (SSE).

```
from openai import OpenAI

client = OpenAI()

stream = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Say this is a test",
        }
    ],
    model="gpt-4o",
    stream=True,
)
for chunk in stream:
    print(chunk.choices[0].delta.content or "", end="")
```

The async client uses the exact same interface.

```
import asyncio
from openai import AsyncOpenAI

client = AsyncOpenAI()


async def main():
    stream = await client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": "Say this 
        stream=True,
    )
    async for chunk in stream:
        print(chunk.choices[0].delta.content or "", end=
asyncio.run(main())
```

Module-level client

> *[!IMPORTANT] We highly recommend instantiating client instances instead of relying on the global client.*

We also expose a global client instance that is accessible in a similar fashion to versions prior to v1.

```
import openai

# optional; defaults to `os.environ['OPENAI_API_KEY']`
openai.api_key = '...'

# all client options can be configured just like the `Ope
openai.base_url = "https://..."
openai.default_headers = {"x-foo": "true"}

completion = openai.chat.completions.create(
    model="gpt-4o",
    messages=[
        {
            "role": "user",
            "content": "How do I output all files in a d
        },
    ],
)
print(completion.choices[0].message.content)
```

The API is the exact same as the standard client instance-based API.

This is intended to be used within REPLs or notebooks for faster iteration, **not** in application code.

We recommend that you always instantiate a client (e.g., with `client = OpenAI()`) in application code because:

- It can be difficult to reason about where client options are configured

- It's not possible to change certain client options without potentially causing race conditions
- It's harder to mock for testing purposes
- It's not possible to control cleanup of network connections

Realtime API beta

The Realtime API enables you to build low-latency, multi-modal conversational experiences. It currently supports text and audio as both input and output, as well as function calling through a WebSocket connection.

Under the hood the SDK uses the `websockets` library to manage connections.

The Realtime API works through a combination of client-sent events and server-sent events. Clients can send events to do things like update session configuration or send text and audio inputs. Server events confirm when audio responses have completed, or when a text response from the model has been received. A full event reference can be found here and a guide can be found here.

Basic text based example:

```
import asyncio
from openai import AsyncOpenAI

async def main():
    client = AsyncOpenAI()

    async with client.beta.realtime.connect(model="gpt-4c
        await connection.session.update(session={'modalit

        await connection.conversation.item.create(
            item={
                "type": "message",
                "role": "user",
                "content": [{"type": "input_text", "text'
            }
        )
        await connection.response.create()
        async for event in connection:
            if event.type == 'response.text.delta':
                print(event.delta, flush=True, end="")

            elif event.type == 'response.text.done':
                print()

            elif event.type == "response.done":
                break

asyncio.run(main())
```

However the real magic of the Realtime API is handling audio inputs / outputs, see this example TUI script for a fully fledged example.

Realtime error handling

Whenever an error occurs, the Realtime API will send an `error` event and the connection will stay open and remain usable. This means you need to handle it yourself, as *no errors are raised directly* by the SDK when an `error` event comes in.

```
client = AsyncOpenAI()

async with client.beta.realtime.connect(model="gpt-4o-rea
    ...
    async for event in connection:
        if event.type == 'error':
            print(event.error.type)
            print(event.error.code)
            print(event.error.event_id)
            print(event.error.message)
```

Using types

Nested request parameters are TypedDicts. Responses are Pydantic models which also provide helper methods for things like:

- Serializing back into JSON, `model.to_json()`
- Converting to a dictionary, `model.to_dict()`

Typed requests and responses provide autocomplete and documentation within your editor. If you would like to see type errors in VS Code to help catch bugs earlier, set `python.analysis.typeCheckingMode` to `basic`.

Pagination

List methods in the OpenAI API are paginated.

This library provides auto-paginating iterators with each list response, so you do not have to request successive pages manually:

```
from openai import OpenAI

client = OpenAI()

all_jobs = []
# Automatically fetches more pages as needed.
for job in client.fine_tuning.jobs.list(
    limit=20,
):
    # Do something with job here
    all_jobs.append(job)
print(all_jobs)
```

Or, asynchronously:

```
import asyncio
from openai import AsyncOpenAI

client = AsyncOpenAI()


async def main() -> None:
    all_jobs = []
    # Iterate through items across all pages, issuing re
    async for job in client.fine_tuning.jobs.list(
        limit=20,
    ):
        all_jobs.append(job)
    print(all_jobs)


asyncio.run(main())
```

Alternatively, you can use the `.has_next_page()`, `.next_page_info()`, or `.get_next_page()` methods for more granular control working with pages:

```
first_page = await client.fine_tuning.jobs.list(
    limit=20,
)
if first_page.has_next_page():
    print(f"will fetch next page using these details: {f
    next_page = await first_page.get_next_page()
    print(f"number of items we just fetched: {len(next_pa

# Remove `await` for non-async usage.
```

Or just work directly with the returned data:

```
first_page = await client.fine_tuning.jobs.list(
    limit=20,
)

print(f"next page cursor: {first_page.after}")  # => "ne
for job in first_page.data:
    print(job.id)

# Remove `await` for non-async usage.
```

Nested params

Nested parameters are dictionaries, typed using `TypedDict`, for example:

```
from openai import OpenAI

client = OpenAI()

completion = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": "Can you generate an example json
        }
    ],
    model="gpt-4o",
    response_format={"type": "json_object"},
)
```

File uploads

Request parameters that correspond to file uploads can be passed as `bytes`, a `PathLike` instance or a tuple of `(filename, contents, media type)`.

```
from pathlib import Path
from openai import OpenAI

client = OpenAI()

client.files.create(
    file=Path("input.jsonl"),
    purpose="fine-tune",
)
```

The async client uses the exact same interface. If you pass a `PathLike` instance, the file contents will be read asynchronously automatically.

Handling errors

When the library is unable to connect to the API (for example, due to network connection problems or a timeout), a subclass of `openai.APIConnectionError` is raised.

When the API returns a non-success status code (that is, 4xx or 5xx response), a subclass of `openai.APIStatusError` is raised, containing `status_code` and `response` properties.

All errors inherit from `openai.APIError`.

```
import openai
from openai import OpenAI

client = OpenAI()

try:
    client.fine_tuning.jobs.create(
        model="gpt-4o",
        training_file="file-abc123",
    )
except openai.APIConnectionError as e:
    print("The server could not be reached")
    print(e.__cause__)  # an underlying Exception, likely
except openai.RateLimitError as e:
    print("A 429 status code was received; we should bac
except openai.APIStatusError as e:
    print("Another non-200-range status code was receive
    print(e.status_code)
    print(e.response)
```

Error codes are as follows:

Status Code	Error Type
400	`BadRequestError`
401	`AuthenticationError`
403	`PermissionDeniedError`
404	`NotFoundError`
422	`UnprocessableEntityError`
429	`RateLimitError`

Status Code	Error Type
>=500	`InternalServerError`
N/A	`APIConnectionError`

Request IDs

> *For more information on debugging requests, see these docs*

All object responses in the SDK provide a `_request_id` property which is added from the `x-request-id` response header so that you can quickly log failing requests and report them back to OpenAI.

```
completion = await client.chat.completions.create(
    messages=[{"role": "user", "content": "Say this is a
)
print(completion._request_id)  # req_123
```

Note that unlike other properties that use an `_` prefix, the `_request_id` property *is* public. Unless documented otherwise, *all* other `_` prefix properties, methods and modules are *private*.

Retries

Certain errors are automatically retried 2 times by default, with a short exponential backoff. Connection errors (for example, due to a network connectivity problem), 408 Request Timeout, 409 Conflict, 429 Rate Limit, and >=500 Internal errors are all retried by default.

You can use the `max_retries` option to configure or disable retry settings:

```
from openai import OpenAI

# Configure the default for all requests:
client = OpenAI(
    # default is 2
    max_retries=0,
)

# Or, configure per-request:
client.with_options(max_retries=5).chat.completions.creat
    messages=[
        {
            "role": "user",
            "content": "How can I get the name of the cu
        }
    ],
    model="gpt-4o",
)
```

Timeouts

By default requests time out after 10 minutes. You can configure this with a `timeout` option, which accepts a float or an `httpx.Timeout` object:

```
from openai import OpenAI

# Configure the default for all requests:
client = OpenAI(
    # 20 seconds (default is 10 minutes)
    timeout=20.0,
)

# More granular control:
client = OpenAI(
    timeout=httpx.Timeout(60.0, read=5.0, write=10.0, co
)

# Override per-request:
client.with_options(timeout=5.0).chat.completions.create
    messages=[
        {
            "role": "user",
            "content": "How can I list all files in a di
        }
    ],
    model="gpt-4o",
)
```

On timeout, an `APITimeoutError` is thrown.

Note that requests that time out are retried twice by default.

Advanced

Logging

We use the standard library `logging` module.

You can enable logging by setting the environment variable `OPENAI_LOG` to `info`.

```
$ export OPENAI_LOG=info
```

Or to `debug` for more verbose logging.

How to tell whether `None` means `null` or missing

In an API response, a field may be explicitly `null`, or missing entirely; in either case, its value is `None` in this library. You can differentiate the two cases with `.model_fields_set`:

```
if response.my_field is None:
  if 'my_field' not in response.model_fields_set:
    print('Got json like {}, without a "my_field" key pre
  else:
    print('Got json like {"my_field": null}.')
```

Accessing raw response data (e.g. headers)

The "raw" Response object can be accessed by prefixing `.with_raw_response.` to any HTTP method call, e.g.,

```
from openai import OpenAI

client = OpenAI()
response = client.chat.completions.with_raw_response.crea
    messages=[{
        "role": "user",
        "content": "Say this is a test",
    }],
    model="gpt-4o",
)
print(response.headers.get('X-My-Header'))

completion = response.parse()  # get the object that `cha
print(completion)
```

These methods return a `LegacyAPIResponse` object. This is a legacy class as we're changing it slightly in the next major version.

For the sync client this will mostly be the same with the exception of `content` & `text` will be methods instead of properties. In the async client, all methods will be async.

A migration script will be provided & the migration in general should be smooth.

`.with_streaming_response`

The above interface eagerly reads the full response body when you make the request, which may not always be what you want.

To stream the response body, use `.with_streaming_response` instead, which requires a context manager and only reads the response body once you call `.read()`, `.text()`, `.json()`, `.iter_bytes()`, `.iter_text()`, `.iter_lines()` or `.parse()`. In the async client, these are async methods.

As such, `.with_streaming_response` methods return a different `APIResponse` object, and the async client returns an `AsyncAPIResponse` object.

```
with client.chat.completions.with_streaming_response.crea
    messages=[
        {
            "role": "user",
            "content": "Say this is a test",
        }
    ],
    model="gpt-4o",
) as response:
    print(response.headers.get("X-My-Header"))

    for line in response.iter_lines():
        print(line)
```

The context manager is required so that the response will reliably be closed.

Making custom/undocumented requests

This library is typed for convenient access to the documented API.

If you need to access undocumented endpoints, params, or response properties, the library can still be used.

Undocumented endpoints

To make requests to undocumented endpoints, you can make requests using `client.get`, `client.post`, and other http verbs. Options on the client will be respected (such as retries) when making this request.

```
import httpx

response = client.post(
    "/foo",
    cast_to=httpx.Response,
    body={"my_param": True},
)
```

```
print(response.headers.get("x-foo"))
```

Undocumented request params

If you want to explicitly send an extra param, you can do so with the `extra_query`, `extra_body`, and `extra_headers` request options.

Undocumented response properties

To access undocumented response properties, you can access the extra fields like `response.unknown_prop`. You can also get all the extra fields on the Pydantic model as a dict with `response.model_extra`.

Configuring the HTTP client

You can directly override the httpx client to customize it for your use case, including:

- Support for proxies
- Custom transports
- Additional advanced functionality

```
import httpx
from openai import OpenAI, DefaultHttpxClient

client = OpenAI(
    # Or use the `OPENAI_BASE_URL` env var
    base_url="http://my.test.server.example.com:8083/v1"
    http_client=DefaultHttpxClient(
        proxy="http://my.test.proxy.example.com",
        transport=httpx.HTTPTransport(local_address="0.0
    ),
)
```

You can also customize the client on a per-request basis by using `with_options()`:

```
client.with_options(http_client=DefaultHttpxClient(...))
```

Managing HTTP resources

By default the library closes underlying HTTP connections whenever the client is garbage collected. You can manually close the client using the `.close()` method if desired, or with a context manager that closes when exiting.

```
from openai import OpenAI

with OpenAI() as client:
  # make requests here
  ...

# HTTP client is now closed
```

Microsoft Azure OpenAI

To use this library with Azure OpenAI, use the `AzureOpenAI` class instead of the `OpenAI` class.

> *[!IMPORTANT] The Azure API shape differs from the core API shape which means that the static types for responses / params won't always be correct.*

```
from openai import AzureOpenAI

# gets the API Key from environment variable AZURE_OPENAI
client = AzureOpenAI(
    # https://learn.microsoft.com/azure/ai-services/opena
    api_version="2023-07-01-preview",
    # https://learn.microsoft.com/azure/cognitive-service
    azure_endpoint="https://example-endpoint.openai.azure
)

completion = client.chat.completions.create(
    model="deployment-name",  # e.g. gpt-35-instant
    messages=[
        {
            "role": "user",
            "content": "How do I output all files in a d
        },
    ],
)
print(completion.to_json())
```

In addition to the options provided in the base `OpenAI` client, the following options are provided:

- `azure_endpoint` (or the `AZURE_OPENAI_ENDPOINT` environment variable)
- `azure_deployment`
- `api_version` (or the `OPENAI_API_VERSION` environment variable)
- `azure_ad_token` (or the `AZURE_OPENAI_AD_TOKEN` environment variable)
- `azure_ad_token_provider`

An example of using the client with Microsoft Entra ID (formerly known as Azure Active Directory) can be found here.

Versioning

This package generally follows SemVer conventions, though certain backwards-incompatible changes may be released as minor versions:

1. Changes that only affect static types, without breaking runtime behavior.
2. Changes to library internals which are technically public but not intended or documented for external use. *(Please open a GitHub issue to let us know if you are relying on such internals.)*
3. Changes that we do not expect to impact the vast majority of users in practice.

We take backwards-compatibility seriously and work hard to ensure you can rely on a smooth upgrade experience.

We are keen for your feedback; please open an issue with questions, bugs, or suggestions.

Determining the installed version

If you've upgraded to the latest version but aren't seeing any new features you were expecting then your python environment is likely still using an older version.

You can determine the version that is being used at runtime with:

```
import openai
print(openai.__version__)
```

Requirements

Python 3.8 or higher.

Contributing

See the contributing documentation.

Contributing to PyPI

Bugs and feedback
Contribute on GitHub
Translate PyPI
Sponsor PyPI
Development credits

Using PyPI

Code of conduct
Report security issue
Privacy Notice
Terms of Use
Acceptable Use Policy

Switch to desktop version

English español français 日本語 português (Brasil) українська Ελληνικά Deutsch 中文 (简体) 中文 (繁體) русский עברית Esperanto 한국어

AWS
Cloud computing and Security Sponsor

Datadog
Monitoring

Fastly
CDN

Google
Download Analytics

Pingdom
Monitoring

Sentry
Error logging

StatusPage
Status page