```
# File generated from our OpenAPI spec by Stainless. See CONTRIBUTING.md for details. from
__future__ import annotations from typing import Dict, List, Type, Union, Iterable, Optional, cast
from functools import partial from typing_extensions import Literal import httpx from .... import
_legacy_response from ...._types import NOT_GIVEN, Body, Query, Headers, NotGiven from
...._utils import maybe_transform, async_maybe_transform from ...._compat import cached_property
from ...._resource import SyncAPIResource, AsyncAPIResource from ...._response import
to_streamed_response_wrapper, async_to_streamed_response_wrapper from ...._streaming import
Stream from ....types.chat import completion_create_params from ...._base_client import
make_request_options from ....lib._parsing import ( ResponseFormatT, validate_input_tools as
_validate_input_tools, parse_chat_completion as _parse_chat_completion,
type_to_response_format_param as _type_to_response_format, ) from ....types.chat_model import
ChatModel from ....lib.streaming.chat import ChatCompletionStreamManager,
AsyncChatCompletionStreamManager from ....types.chat.chat_completion import ChatCompletion
from ....types.chat.chat_completion_chunk import ChatCompletionChunk from
....types.chat.parsed_chat_completion import ParsedChatCompletion from
....types.chat.chat_completion_modality import ChatCompletionModality from
....types.chat.chat_completion_tool_param import ChatCompletionToolParam from
....types.chat.chat_completion_audio_param import ChatCompletionAudioParam from
....types.chat.chat_completion_message_param import ChatCompletionMessageParam from
....types.chat.chat_completion_stream_options_param import ChatCompletionStreamOptionsParam
from ....types.chat.chat_completion_prediction_content_param import
ChatCompletionPredictionContentParam from
....types.chat.chat_completion_tool_choice_option_param import
ChatCompletionToolChoiceOptionParam __all__ = ["Completions", "AsyncCompletions"] class
Completions(SyncAPIResource): @cached_property def with_raw_response(self) ->
CompletionsWithRawResponse: """ This property can be used as a prefix for any HTTP method call
to return the the raw response object instead of the parsed content. For more information, see
```

https://www.github.com/openai/openai-python#accessing-raw-response-data-eg-headers """ return CompletionsWithRawResponse(self) @cached_property def with_streaming_response(self) -> CompletionsWithStreamingResponse: """ An alternative to `.with_raw_response` that doesn't eagerly read the response body. For more information, see https://www.github.com/openai/openai-python#with_streaming_response """ return CompletionsWithStreamingResponse(self) def parse(self, *, messages: Iterable[ChatCompletionMessageParam], model: Union[str, ChatModel], audio: Optional[ChatCompletionAudioParam] | NotGiven = NOT_GIVEN, response_format: type[ResponseFormatT] | NotGiven = NOT_GIVEN, frequency_penalty: Optional[float] | NotGiven = NOT_GIVEN, function_call: completion_create_params.FunctionCall | NotGiven = NOT_GIVEN, functions: Iterable[completion_create_params.Function] | NotGiven = NOT_GIVEN, logit_bias: Optional[Dict[str, int]] | NotGiven = NOT_GIVEN, logprobs: Optional[bool] | NotGiven = NOT_GIVEN, max_completion_tokens: Optional[int] | NotGiven = NOT_GIVEN, max_tokens: Optional[int] | NotGiven = NOT_GIVEN, metadata: Optional[Dict[str, str]] | NotGiven = NOT_GIVEN, modalities: Optional[List[ChatCompletionModality]] | NotGiven = NOT_GIVEN, n: Optional[int] | NotGiven = NOT_GIVEN, parallel_tool_calls: bool | NotGiven = NOT_GIVEN, prediction: Optional[ChatCompletionPredictionContentParam] | NotGiven = NOT_GIVEN, presence_penalty: Optional[float] | NotGiven = NOT_GIVEN, seed: Optional[int] | NotGiven = NOT_GIVEN, service_tier: Optional[Literal["auto", "default"]] | NotGiven = NOT_GIVEN, stop: Union[Optional[str], List[str]] | NotGiven = NOT_GIVEN, store: Optional[bool] | NotGiven = NOT_GIVEN, stream_options: Optional[ChatCompletionStreamOptionsParam] | NotGiven = NOT_GIVEN, temperature: Optional[float] | NotGiven = NOT_GIVEN, tool_choice: ChatCompletionToolChoiceOptionParam | NotGiven = NOT_GIVEN, tools: Iterable[ChatCompletionToolParam] | NotGiven = NOT_GIVEN, top_logprobs: Optional[int] | NotGiven = NOT_GIVEN, top_p: Optional[float] | NotGiven = NOT_GIVEN, user: str | NotGiven = NOT_GIVEN, # Use the following arguments if you need to pass additional parameters to the API that aren't available via kwargs. # The extra values given here take precedence over values defined on the client or passed to this method. extra_headers: Headers | None = None, extra_query: Query | None = None, extra_body: Body | None = None, timeout: float | httpx.Timeout | None | NotGiven = NOT_GIVEN,) -> ParsedChatCompletion[ResponseFormatT]: """Wrapper over the `client.chat.completions.create()` method that provides richer integrations with Python specific types & returns a `ParsedChatCompletion` object, which is a subclass of the standard `ChatCompletion` class. You can pass a pydantic model to this method and it will automatically convert the model into a JSON schema, send it to the API and parse the response content back into the given model. This method will also automatically parse `function` tool calls if: \- You use the `openai.pydantic_function_tool()` helper method \- You mark your tool schema with `"strict": True` Example usage: ```py from pydantic import BaseModel from openai import OpenAI class Step(BaseModel): explanation: str output: str class MathResponse(BaseModel): steps: List[Step] final_answer: str client = OpenAI() completion = client.beta.chat.completions.parse(model="gpt-4o-2024-08-06", messages=[{"role": "system", "content": "You are a helpful math tutor."}, {"role": "user", "content": "solve 8x + 31 = 2"},], response_format=MathResponse,) message = completion.choices[0].message if message.parsed: print(message.parsed.steps) print("answer: ", message.parsed.final_answer) ``` """ _validate_input_tools(tools) extra_headers = { "X-Stainless-Helper-Method": "beta.chat.completions.parse", **(extra_headers or {}), } def parser(raw_completion: ChatCompletion) -> ParsedChatCompletion[ResponseFormatT]: return _parse_chat_completion(response_format=response_format, chat_completion=raw_completion, input_tools=tools,) return self._post("/chat/completions", body=maybe_transform({ "messages": messages, "model": model, "audio": audio, "frequency_penalty": frequency_penalty, "function_call": function_call, "functions": functions, "logit_bias": logit_bias, "logprobs": logprobs, "max_completion_tokens": max_completion_tokens, "max_tokens": max_tokens, "metadata": metadata, "modalities": modalities, "n": n, "parallel_tool_calls": parallel_tool_calls, "prediction": prediction, "presence_penalty": presence_penalty, "response_format": _type_to_response_format(response_format), "seed": seed, "service_tier": service_tier, "stop": stop, "store": store, "stream": False, "stream_options": stream_options, "temperature": temperature, "tool_choice": tool_choice, "tools": tools, "top_logprobs": top_logprobs, "top_p": top_p, "user": user, }, completion_create_params.CompletionCreateParams,), options=make_request_options(extra_headers=extra_headers, extra_query=extra_query, extra_body=extra_body, timeout=timeout, post_parser=parser,), # we turn the `ChatCompletion` instance into a `ParsedChatCompletion` # in the `parser` function above cast_to=cast(Type[ParsedChatCompletion[ResponseFormatT]], ChatCompletion), stream=False,) def stream(self, *, messages: Iterable[ChatCompletionMessageParam], model: Union[str, ChatModel], audio: Optional[ChatCompletionAudioParam] | NotGiven = NOT_GIVEN, response_format: completion_create_params.ResponseFormat | type[ResponseFormatT] | NotGiven = NOT_GIVEN, frequency_penalty: Optional[float] | NotGiven = NOT_GIVEN, function_call: completion_create_params.FunctionCall | NotGiven = NOT_GIVEN, functions: Iterable[completion_create_params.Function] | NotGiven = NOT_GIVEN, logit_bias: Optional[Dict[str, int]] | NotGiven = NOT_GIVEN, logprobs: Optional[bool] | NotGiven = NOT_GIVEN, max_completion_tokens: Optional[int] | NotGiven = NOT_GIVEN, max_tokens: Optional[int] | NotGiven = NOT_GIVEN, metadata: Optional[Dict[str, str]] | NotGiven = NOT_GIVEN, modalities: Optional[List[ChatCompletionModality]] | NotGiven = NOT_GIVEN, n: Optional[int] | NotGiven = NOT_GIVEN, parallel_tool_calls: bool | NotGiven = NOT_GIVEN, prediction: Optional[ChatCompletionPredictionContentParam] | NotGiven = NOT_GIVEN, presence_penalty: Optional[float] | NotGiven = NOT_GIVEN, seed: Optional[int] | NotGiven = NOT_GIVEN, service_tier: Optional[Literal["auto", "default"]] | NotGiven = NOT_GIVEN, stop: Union[Optional[str], List[str]] | NotGiven = NOT_GIVEN, store: Optional[bool] | NotGiven = NOT_GIVEN, stream_options: Optional[ChatCompletionStreamOptionsParam] | NotGiven = NOT_GIVEN, temperature: Optional[float] | NotGiven = NOT_GIVEN, tool_choice: ChatCompletionToolChoiceOptionParam | NotGiven = NOT_GIVEN, tools: Iterable[ChatCompletionToolParam] | NotGiven = NOT_GIVEN, top_logprobs: Optional[int] | NotGiven = NOT_GIVEN, top_p: Optional[float] | NotGiven = NOT_GIVEN, user: str | NotGiven = NOT_GIVEN, # Use the following arguments if you need to pass additional parameters to the API

that aren't available via kwargs. # The extra values given here take precedence over values defined on the client or passed to this method. extra_headers: Headers | None = None, extra_query: Query | None = None, extra_body: Body | None = None, timeout: float | httpx.Timeout | None | NotGiven = NOT_GIVEN,) -> ChatCompletionStreamManager[ResponseFormatT]: """Wrapper over the `client.chat.completions.create(stream=True)` method that provides a more granular event API and automatic accumulation of each delta. This also supports all of the parsing utilities that `.parse()` does. Unlike `.create(stream=True)`, the `.stream()` method requires usage within a context manager to prevent accidental leakage of the response: ```py with client.beta.chat.completions.stream(model="gpt-4o-2024-08-06", messages=[...],) as stream: for event in stream: if event.type == "content.delta": print(event.delta, flush=True, end="") ``` When the context manager is entered, a `ChatCompletionStream` instance is returned which, like `.create(stream=True)` is an iterator. The full list of events that are yielded by the iterator are outlined in [these docs](https://github.com/openai/openai-python/blob/main/helpers.md#chat-completions-events).
When the context manager exits, the response will be closed, however the `stream` instance is still available outside the context manager. """ extra_headers = { "X-Stainless-Helper-Method": "beta.chat.completions.stream", **(extra_headers or {}), } api_request: partial[Stream[ChatCompletionChunk]] = partial(self._client.chat.completions.create, messages=messages, model=model, audio=audio, stream=True, response_format=_type_to_response_format(response_format), frequency_penalty=frequency_penalty, function_call=function_call, functions=functions, logit_bias=logit_bias, logprobs=logprobs, max_completion_tokens=max_completion_tokens, max_tokens=max_tokens, metadata=metadata, modalities=modalities, n=n, parallel_tool_calls=parallel_tool_calls, prediction=prediction, presence_penalty=presence_penalty, seed=seed, service_tier=service_tier, store=store, stop=stop, stream_options=stream_options, temperature=temperature, tool_choice=tool_choice, tools=tools, top_logprobs=top_logprobs, top_p=top_p, user=user, extra_headers=extra_headers, extra_query=extra_query,

extra_body=extra_body, timeout=timeout,) return ChatCompletionStreamManager(api_request, response_format=response_format, input_tools=tools,) class AsyncCompletions(AsyncAPIResource): @cached_property def with_raw_response(self) -> AsyncCompletionsWithRawResponse: """ This property can be used as a prefix for any HTTP method call to return the the raw response object instead of the parsed content. For more information, see https://www.github.com/openai/openai-python#accessing-raw-response-data-eg-headers """ return AsyncCompletionsWithRawResponse(self) @cached_property def with_streaming_response(self) -> AsyncCompletionsWithStreamingResponse: """ An alternative to `.with_raw_response` that doesn't eagerly read the response body. For more information, see https://www.github.com/openai/openai-python#with_streaming_response """ return AsyncCompletionsWithStreamingResponse(self) async def parse(self, *, messages: Iterable[ChatCompletionMessageParam], model: Union[str, ChatModel], audio: Optional[ChatCompletionAudioParam] | NotGiven = NOT_GIVEN, response_format: type[ResponseFormatT] | NotGiven = NOT_GIVEN, frequency_penalty: Optional[float] | NotGiven = NOT_GIVEN, function_call: completion_create_params.FunctionCall | NotGiven = NOT_GIVEN, functions: Iterable[completion_create_params.Function] | NotGiven = NOT_GIVEN, logit_bias: Optional[Dict[str, int]] | NotGiven = NOT_GIVEN, logprobs: Optional[bool] | NotGiven = NOT_GIVEN, max_completion_tokens: Optional[int] | NotGiven = NOT_GIVEN, max_tokens: Optional[int] | NotGiven = NOT_GIVEN, metadata: Optional[Dict[str, str]] | NotGiven = NOT_GIVEN, modalities: Optional[List[ChatCompletionModality]] | NotGiven = NOT_GIVEN, n: Optional[int] | NotGiven = NOT_GIVEN, parallel_tool_calls: bool | NotGiven = NOT_GIVEN, prediction: Optional[ChatCompletionPredictionContentParam] | NotGiven = NOT_GIVEN, presence_penalty: Optional[float] | NotGiven = NOT_GIVEN, seed: Optional[int] | NotGiven = NOT_GIVEN, service_tier: Optional[Literal["auto", "default"]] | NotGiven = NOT_GIVEN, stop: Union[Optional[str], List[str]] | NotGiven = NOT_GIVEN, store: Optional[bool] | NotGiven = NOT_GIVEN, stream_options: Optional[ChatCompletionStreamOptionsParam] | NotGiven = NOT_GIVEN, temperature: Optional[float] | NotGiven = NOT_GIVEN, tool_choice: ChatCompletionToolChoiceOptionParam | NotGiven = NOT_GIVEN, tools: Iterable[ChatCompletionToolParam] | NotGiven = NOT_GIVEN, top_logprobs: Optional[int] | NotGiven = NOT_GIVEN, top_p: Optional[float] | NotGiven = NOT_GIVEN, user: str | NotGiven = NOT_GIVEN, # Use the following arguments if you need to pass additional parameters to the API that aren't available via kwargs. # The extra values given here take precedence over values defined on the client or passed to this method. extra_headers: Headers | None = None, extra_query: Query | None = None, extra_body: Body | None = None, timeout: float | httpx.Timeout | None | NotGiven = NOT_GIVEN,) -> ParsedChatCompletion[ResponseFormatT]: """Wrapper over the `client.chat.completions.create()` method that provides richer integrations with Python specific types & returns a `ParsedChatCompletion` object, which is a subclass of the standard `ChatCompletion` class. You can pass a pydantic model to this method and it will automatically convert the model into a JSON schema, send it to the API and parse the response content back into the given model. This method will also automatically parse `function` tool calls if: \- You use the `openai.pydantic_function_tool()` helper method \- You mark your tool schema with `"strict": True` Example usage: ```py from pydantic import BaseModel from openai import AsyncOpenAI class Step(BaseModel): explanation: str output: str class MathResponse(BaseModel): steps: List[Step] final_answer: str client = AsyncOpenAI() completion = await client.beta.chat.completions.parse(model="gpt-4o-2024-08-06", messages=[{"role": "system", "content": "You are a helpful math tutor."}, {"role": "user", "content": "solve 8x + 31 = 2"},], response_format=MathResponse,) message = completion.choices[0].message if message.parsed: print(message.parsed.steps) print("answer: ", message.parsed.final_answer) ``` """ _validate_input_tools(tools) extra_headers = { "X-Stainless-Helper-Method": "beta.chat.completions.parse", **(extra_headers or {}), } def parser(raw_completion: ChatCompletion) -> ParsedChatCompletion[ResponseFormatT]: return _parse_chat_completion(response_format=response_format, chat_completion=raw_completion, input_tools=tools,) return await self._post("/chat/completions", body=await async_maybe_transform({ "messages": messages, "model": model, "audio": audio, "frequency_penalty": frequency_penalty, "function_call": function_call, "functions": functions, "logit_bias": logit_bias, "logprobs": logprobs, "max_completion_tokens": max_completion_tokens, "max_tokens": max_tokens, "metadata": metadata, "modalities": modalities, "n": n, "parallel_tool_calls": parallel_tool_calls, "prediction": prediction, "presence_penalty": presence_penalty, "response_format": _type_to_response_format(response_format), "seed": seed, "service_tier": service_tier, "store": store, "stop": stop, "stream": False, "stream_options": stream_options, "temperature": temperature, "tool_choice": tool_choice, "tools": tools, "top_logprobs": top_logprobs, "top_p": top_p, "user": user, }, completion_create_params.CompletionCreateParams,), options=make_request_options(extra_headers=extra_headers, extra_query=extra_query, extra_body=extra_body, timeout=timeout, post_parser=parser,), # we turn the `ChatCompletion` instance into a `ParsedChatCompletion` # in the `parser` function above cast_to=cast(Type[ParsedChatCompletion[ResponseFormatT]], ChatCompletion), stream=False,) def stream(self, *, messages: Iterable[ChatCompletionMessageParam], model: Union[str, ChatModel], audio: Optional[ChatCompletionAudioParam] | NotGiven = NOT_GIVEN, response_format: completion_create_params.ResponseFormat | type[ResponseFormatT] | NotGiven = NOT_GIVEN, frequency_penalty: Optional[float] | NotGiven = NOT_GIVEN, function_call: completion_create_params.FunctionCall | NotGiven = NOT_GIVEN, functions: Iterable[completion_create_params.Function] | NotGiven = NOT_GIVEN, logit_bias: Optional[Dict[str, int]] | NotGiven = NOT_GIVEN, logprobs: Optional[bool] | NotGiven = NOT_GIVEN, max_completion_tokens: Optional[int] | NotGiven = NOT_GIVEN, max_tokens: Optional[int] | NotGiven = NOT_GIVEN, metadata: Optional[Dict[str, str]] | NotGiven = NOT_GIVEN, modalities: Optional[List[ChatCompletionModality]] | NotGiven = NOT_GIVEN, n: Optional[int] | NotGiven = NOT_GIVEN, parallel_tool_calls: bool | NotGiven = NOT_GIVEN, prediction: Optional[ChatCompletionPredictionContentParam] | NotGiven = NOT_GIVEN, presence_penalty: Optional[float] | NotGiven = NOT_GIVEN, seed: Optional[int] | NotGiven = NOT_GIVEN, service_tier: Optional[Literal["auto", "default"]] | NotGiven = NOT_GIVEN, stop: Union[Optional[str], List[str]] | NotGiven = NOT_GIVEN, store: Optional[bool] | NotGiven = NOT_GIVEN, stream_options: Optional[ChatCompletionStreamOptionsParam] | NotGiven = NOT_GIVEN, temperature: Optional[float] | NotGiven = NOT_GIVEN, tool_choice: ChatCompletionToolChoiceOptionParam | NotGiven = NOT_GIVEN, tools: Iterable[ChatCompletionToolParam] | NotGiven = NOT_GIVEN, top_logprobs: Optional[int] | NotGiven = NOT_GIVEN, top_p: Optional[float] | NotGiven = NOT_GIVEN, user: str | NotGiven = NOT_GIVEN, # Use the following arguments if you need to pass additional parameters to the API that aren't available via kwargs. # The extra values given here take precedence over values defined on the client or passed to this method. extra_headers: Headers | None = None, extra_query: Query | None = None, extra_body: Body | None = None, timeout: float | httpx.Timeout | None | NotGiven = NOT_GIVEN,) -> AsyncChatCompletionStreamManager[ResponseFormatT]: """Wrapper over the `client.chat.completions.create(stream=True)` method that provides a more granular event API and automatic accumulation of each delta. This also supports all of the parsing utilities that `.parse()` does. Unlike `.create(stream=True)`, the `.stream()` method requires usage within a context manager to prevent accidental leakage of the response: ```py async with client.beta.chat.completions.stream(model="gpt-4o-2024-08-06", messages=[...],) as stream: async for event in stream: if event.type == "content.delta": print(event.delta, flush=True, end="") ``` When the context manager is entered, an `AsyncChatCompletionStream` instance is returned which, like `.create(stream=True)` is an async iterator. The full list of events that are yielded by the iterator are outlined in [these docs](https://github.com/openai/openai-python/blob/main/helpers.md#chat-completions-events).

When the context manager exits, the response will be closed, however the `stream` instance is still available outside the context manager. """ _validate_input_tools(tools) extra_headers = { "X-Stainless-Helper-Method": "beta.chat.completions.stream", **(extra_headers or {}), } api_request = self._client.chat.completions.create(messages=messages, model=model, audio=audio, stream=True, response_format=_type_to_response_format(response_format), frequency_penalty=frequency_penalty, function_call=function_call, functions=functions,

```
logit_bias=logit_bias, logprobs=logprobs, max_completion_tokens=max_completion_tokens,
max_tokens=max_tokens, metadata=metadata, modalities=modalities, n=n,
parallel_tool_calls=parallel_tool_calls, prediction=prediction, presence_penalty=presence_penalty,
seed=seed, service_tier=service_tier, stop=stop, store=store, stream_options=stream_options,
temperature=temperature, tool_choice=tool_choice, tools=tools, top_logprobs=top_logprobs,
top_p=top_p, user=user, extra_headers=extra_headers, extra_query=extra_query,
extra_body=extra_body, timeout=timeout, ) return AsyncChatCompletionStreamManager(
api_request, response_format=response_format, input_tools=tools, ) class
CompletionsWithRawResponse: def __init__(self, completions: Completions) -> None:
self._completions = completions self.parse = _legacy_response.to_raw_response_wrapper(
completions.parse, ) class AsyncCompletionsWithRawResponse: def __init__(self, completions:
AsyncCompletions) -> None: self._completions = completions self.parse =
_legacy_response.async_to_raw_response_wrapper( completions.parse, ) class
CompletionsWithStreamingResponse: def __init__(self, completions: Completions) -> None:
self._completions = completions self.parse = to_streamed_response_wrapper( completions.parse, )
class AsyncCompletionsWithStreamingResponse: def __init__(self, completions:
AsyncCompletions) -> None: self._completions = completions self.parse =
async_to_streamed_response_wrapper( completions.parse, )
```