On this page

oai.client

OpenAIWrapper Objects

```
class OpenAIWrapper()
```

A wrapper class for openai client.

__init__

```
def __init__(*,
             config_list: Optional[List[Dict[str, Any]]] = None,
             **base_config: Any)
```

Arguments:

- `config_list` - a list of config dicts to override the base_config. They can contain additional kwargs as allowed in the create method. E.g.,

```
config_list=[
    {
        "model": "gpt-4",
        "api_key": os.environ.get("AZURE_OPENAI_API_KEY"),
        "api_type": "azure",
        "base_url": os.environ.get("AZURE_OPENAI_API_BASE"),
        "api_version": "2023-03-15-preview",
    },
    {
        "model": "gpt-3.5-turbo",
        "api_key": os.environ.get("OPENAI_API_KEY"),
        "api_type": "open_ai",
        "base_url": "https://api.openai.com/v1",
    },
    {
        "model": "llama-7B",
        "base_url": "http://127.0.0.1:8080",
        "api_type": "open_ai",
    }
]
```

- `base_config` - base config. It can contain both keyword arguments for openai client and additional kwargs.

create

```
def create(**config: Any) -> ChatCompletion
```

Make a completion for a given config using openai's clients. Besides the kwargs allowed in openai's client, we allow the following additional kwargs. The config in each client will be overridden by the config.

Arguments:

- context (Dict | None): The context to instantiate the prompt or messages. Default to None. It needs to contain keys that are used by the prompt template or the filter function. E.g., `prompt="Complete the following sentence: {prefix}, context={"prefix": "Today I feel"}`. The actual prompt will be: "Complete the following sentence: Today I feel". More examples can be found at templating.
- `cache_seed` (int | None) for the cache. Default to 41. An integer cache_seed is useful when implementing "controlled randomness" for the completion. None for no caching.
- filter_func (Callable | None): A function that takes in the context and the response and returns a boolean to indicate whether the response is valid. E.g.,

```
def yes_or_no_filter(context, response):
    return context.get("yes_or_no_choice", False) is False or any(
        text in ["Yes.", "No."] for text in client.extract_text_or_completion_object(response)
    )
```

- allow_format_str_template (bool | None): Whether to allow format string template in the config. Default to false.
- api_version (str | None): The api version. Default to None. E.g., "2023-08-01-preview".

print_usage_summary

```
def print_usage_summary(
        mode: Union[str, List[str]] = ["actual", "total"]) -> None
```

Print the usage summary.

clear_usage_summary

```
def clear_usage_summary() -> None
```

Clear the usage summary.

cost

```
def cost(response: Union[ChatCompletion, Completion]) -> float
```

Calculate the cost of the response.

extract_text_or_completion_object

```
@classmethod
def extract_text_or_completion_object(
    cls, response: Union[ChatCompletion, Completion]
) -> Union[List[str], List[ChatCompletionMessage]]
```

Extract the text or ChatCompletion objects from a completion or chat response.

Arguments:

- `response` *ChatCompletion* | *Completion* - The response from openai.

Returns:

A list of text, or a list of ChatCompletion objects if function_call/tool_calls are present.

Edit this page

Previous
« user_proxy_agent

Next
completion »

Community

Discord

Twitter