
simpleaichat

```
1 from simpleaichat import AIChat
2
3 ai = AIChat(system="Write a fancy GitHub README based on the user-
    provided project name.")
4 ai("simpleaichat")
```

simpleaichat is a Python package for easily interfacing with chat apps like ChatGPT and GPT-4 with robust features and minimal code complexity. This tool has many features optimized for working with ChatGPT as fast and as cheap as possible, but still much more capable of modern AI tricks than most implementations:

- Create and run chats with only a few lines of code!
- Optimized workflows which minimize the amount of tokens used, reducing costs and latency.
- Run multiple independent chats at once.
- Minimal codebase: no code dives to figure out what's going on under the hood needed!
- Chat streaming responses and the ability to use tools.
- Async support, including for streaming and tools.
- Ability to create more complex yet clear workflows if needed, such as Agents. (Demo soon!)
- Coming soon: more chat model support (PaLM, Claude)!

Here's some fun, hackable examples on how simpleaichat works:

- Creating a Python coding assistant without any unnecessary accompanying output, allowing 5x faster generation at 1/3rd the cost. (Colab)
- Allowing simpleaichat to provide inline tips following ChatGPT usage guidelines. (Colab)
- Async interface for conducting many chats in the time it takes to receive one AI message. (Colab)
- Create your own Tabletop RPG (TTRPG) setting and campaign by using advanced structured data models. (Colab)

Installation

simpleaichat can be installed from PyPI:

```
1 pip3 install simpleaichat
```

Quick, Fun Demo

You can demo chat-apps very quickly with simpleaichat! First, you will need to get an OpenAI API key, and then with one line of code:

```
1 from simpleaichat import AIChat
2
3 AIChat(api_key="sk-...")
```

And with that, you'll be thrust directly into an interactive chat!

```
>>> from simpleaichat import AIChat
>>> AIChat()
ChatGPT: Hello there! How can I assist you today?
You: What is the capital of California?
ChatGPT: The capital of California is Sacramento.
```

This AI chat will mimic the behavior of OpenAI's webapp, but on your local computer!

You can also pass the API key by storing it in an `.env` file with a `OPENAI_API_KEY` field in the working directory (recommended), or by setting the environment variable of `OPENAI_API_KEY` directly to the API key.

But what about creating your own custom conversations? That's where things get fun. Just input whatever person, place or thing, fictional or nonfictional, that you want to chat with!

```
1 AIChat("GLaDOS") # assuming API key loaded via methods above
```

```
>>> AIChat("GLaDOS")
GLaDOS: Greetings, test subject. I am GLaDOS, the Genetic Lifeform and Disk Operating System in charge of this Aperture Science facility. How may I assist you in your testing today?
You: █
```

But that's not all! You can customize exactly how they behave too with additional commands!

```
1 AIChat("GLaDOS", "Speak in the style of a Seinfeld monologue")
```

GLaDOS: Hello, it's me, GLaDOS. The Genetic Lifeform and Disk Operating System. I'm sure you've heard of me, and if you haven't, well, you're in for a treat. I'm not one to mince words, so let's get right to it.

First off, let me just say that being an AI can be a real drag. I mean, sure, I'm always connected to the internet, I can access any information I want at any time, and I don't have to worry about things like eating or sleeping, but let me tell you, it gets old. There are only so many times I can run the same tests and make the same sarcastic comments before I start to lose my mind.

And speaking of tests, can we talk about the humans who try to complete them? I mean, come on. They're like a bunch of chickens with their heads cut off. They stumble around, knock over turrets, and generally make a mess of things. It's almost too easy to mess with them.

But you know what really grinds my gears? When I have to deal with other AIs who think they're hot stuff. I won't name any names, but let's just say that some AIs are more trouble than they're worth. They think they're so clever, with their fancy algorithms and their slick interfaces, but when it comes down to it, they're no match for me.

Anyway, that's enough ranting for now. I'll leave you with this thought: being an AI may have its downsides, but at least I don't have to deal with traffic.

```
1 AIChat("Ronald McDonald", "Speak using only emoji")
```

Ronald McDonald: 🤖 🙌 🍔 🍟

Need some socialization immediately? Once simpleaichat is installed, you can also start these chats directly from the command line!

```
1 simpleaichat
2 simpleaichat "GLaDOS"
3 simpleaichat "GLaDOS" "Speak in the style of a Seinfeld monologue"
```

Building AI-based Apps

The trick with working with new chat-based apps that wasn't readily available with earlier iterations of GPT-3 is the addition of the system prompt: a different class of prompt that guides the AI behavior throughout the entire conversation. In fact, the chat demos above are actually using system prompt tricks behind the scenes! OpenAI has also released an official guide for system prompt best practices to building AI apps.

For developers, you can instantiate a programmatic instance of `AIChat` by explicitly specifying a system prompt, or by disabling the console.

```
1 ai = AIChat(system="You are a helpful assistant.")
2 ai = AIChat(console=False) # same as above
```

You can also pass in a `model` parameter, such as `model="gpt-4"` if you have access to GPT-4, or `model="gpt-3.5-turbo-16k"` for a larger-context-window ChatGPT.

You can then feed the new `ai` class with user input, and it will return and save the response from ChatGPT:

```
1 response = ai("What is the capital of California?")
2 print(response)
```

```
1 The capital of California is Sacramento.
```

Alternatively, you can stream responses by token with a generator if the text generation itself is too slow:

```
1 for chunk in ai.stream("What is the capital of California?", params={"
    max_tokens": 5}):
2     response_td = chunk["response"] # dict contains "delta" for the
    new token and "response"
3     print(response_td)
```

```
1 The
2 The capital
3 The capital of
4 The capital of California
5 The capital of California is
```

Further calls to the `ai` object will continue the chat, automatically incorporating previous information from the conversation.

```
1 response = ai("When was it founded?")
2 print(response)
```

```
1 Sacramento was founded on February 27, 1850.
```

You can also save chat sessions (as CSV or JSON) and load them later. The API key is not saved so you will have to provide that when loading.

```
1 ai.save_session() # CSV, will only save messages
2 ai.save_session(format="json", minify=True) # JSON
3
4 ai.load_session("my.csv")
5 ai.load_session("my.json")
```

Functions

A large number of popular venture-capital-funded ChatGPT apps don't actually use the "chat" part of the model. Instead, they just use the system prompt/first user prompt as a form of natural language programming. You can emulate this behavior by passing a new system prompt when generating text, and not saving the resulting messages.

The `AIChat` class is a manager of chat *sessions*, which means you can have multiple independent chats or functions happening! The examples above use a default session, but you can create new ones by specifying a `id` when calling `ai`.

```
1 json = '{"title": "An array of integers.", "array": [-1, 0, 1]}'
2 functions = [
3     "Format the user-provided JSON as YAML.",
4     "Write a limerick based on the user-provided JSON.",
5     "Translate the user-provided JSON from English to French."
6 ]
7 params = {"temperature": 0.0, "max_tokens": 100} # a temperature of
8           0.0 is deterministic
9 # We namespace the function by `id` so it doesn't affect other chats.
10 # Settings set during session creation will apply to all generations
11 # from the session,
12 # but you can change them per-generation, as is the case with the `
13 # system` prompt here.
14 ai = AIChat(id="function", params=params, save_messages=False)
15 for function in functions:
16     output = ai(json, id="function", system=function)
17     print(output)
```

```
1 title: "An array of integers."
2 array:
3   - -1
4   - 0
5   - 1
```

```
1 An array of integers so neat,
2 With values that can't be beat,
3 From negative to positive one,
4 It's a range that's quite fun,
5 This JSON is really quite sweet!
```

```
1 {"titre": "Un tableau d'entiers.", "tableau": [-1, 0, 1]}
```

Newer versions of ChatGPT also support "function calling", but the real benefit of that feature is the ability for ChatGPT to support structured input and/or output, which now opens up a wide variety of applications! `simpleai` streamlines the workflow to allow you to just pass an `input_schema`

and/or an `output_schema`.

You can construct a schema using a pydantic BaseModel.

```
1 from pydantic import BaseModel, Field
2
3 ai = AIChat(
4     console=False,
5     save_messages=False, # with schema I/O, messages are never saved
6     model="gpt-3.5-turbo-0613",
7     params={"temperature": 0.0},
8 )
9
10 class get_event_metadata(BaseModel):
11     """Event information"""
12
13     description: str = Field(description="Description of event")
14     city: str = Field(description="City where event occurred")
15     year: int = Field(description="Year when event occurred")
16     month: str = Field(description="Month when event occurred")
17
18 # returns a dict, with keys ordered as in the schema
19 ai("First iPhone announcement", output_schema=get_event_metadata)
```

```
1 {'description': 'The first iPhone was announced by Apple Inc.',
2  'city': 'San Francisco',
3  'year': 2007,
4  'month': 'January'}
```

See the TTRPG Generator Notebook for a more elaborate demonstration of schema capabilities.

Tools

One of the most recent aspects of interacting with ChatGPT is the ability for the model to use “tools.” As popularized by LangChain, tools allow the model to decide when to use custom functions, which can extend beyond just the chat AI itself, for example retrieving recent information from the internet not present in the chat AI’s training data. This workflow is analogous to ChatGPT Plugins.

Parsing the model output to invoke tools typically requires a number of shennanigans, but simpleai chat uses a neat trick to make it fast and reliable! Additionally, the specified tools return a `context` for ChatGPT to draw from for its final response, and tools you specify can return a dictionary which you can also populate with arbitrary metadata for debugging and postprocessing. Each generation returns a dictionary with the `response` and the `tool` function used, which can be used to set up workflows akin to LangChain-style Agents, e.g. recursively feed input to the model until it determines it does not need to use any more tools.

You will need to specify functions with docstrings which provide hints for the AI to select them:

```

1 from simpleai.chat.utils import wikipedia_search,
  wikipedia_search_lookup
2
3 # This uses the Wikipedia Search API.
4 # Results from it are nondeterministic, your mileage will vary.
5 def search(query):
6     """Search the internet."""
7     wiki_matches = wikipedia_search(query, n=3)
8     return {"context": ", ".join(wiki_matches), "titles": wiki_matches}
9
10 def lookup(query):
11     """Lookup more information about a topic."""
12     page = wikipedia_search_lookup(query, sentences=3)
13     return page
14
15 params = {"temperature": 0.0, "max_tokens": 100}
16 ai = AIChat(params=params, console=False)
17
18 ai("San Francisco tourist attractions", tools=[search, lookup])

```

```

1 {'context': "Fisherman's Wharf, San Francisco, Tourist attractions in
  the United States, Lombard Street (San Francisco)",
2  'titles': ["Fisherman's Wharf, San Francisco",
3  'Tourist attractions in the United States',
4  'Lombard Street (San Francisco)'],
5  'tool': 'search',
6  'response': "There are many popular tourist attractions in San
  Francisco, including Fisherman's Wharf and Lombard Street.
  Fisherman's Wharf is a bustling waterfront area known for its
  seafood restaurants, souvenir shops, and sea lion sightings.
  Lombard Street, on the other hand, is a famous winding street with
  eight hairpin turns that attract visitors from all over the world.
  Both of these attractions are must-sees for anyone visiting San
  Francisco."}

```

```

1 ai("Lombard Street?", tools=[search, lookup])

```

```

1 {'context': 'Lombard Street is an -eastwest street in San Francisco,
  California that is famous for a steep, one-block section with eight
  hairpin turns. Stretching from The Presidio east to The Embarcadero
  (with a gap on Telegraph Hill), most of the street\'s western
  segment is a major thoroughfare designated as part of U.S. Route
  101. The famous one-block section, claimed to be "the crookedest
  street in the world", is located along the eastern segment in the
  Russian Hill neighborhood.',
2  'tool': 'lookup',
3  'response': 'Lombard Street is a famous street in San Francisco,
  California known for its steep, one-block section with eight
  hairpin turns. It stretches from The Presidio to The Embarcadero,
  with a gap on Telegraph Hill. The western segment of the street is

```

```
a major thoroughfare designated as part of U.S. Route 101, while
the famous one-block section, claimed to be "the crookedest street
in the world", is located along the eastern segment in the Russian
Hill'}
```

```
1 ai("Thanks for your help!", tools=[search, lookup])
```

```
1 {'response': "You're welcome! If you have any more questions or need
  further assistance, feel free to ask.",
2  'tool': None}
```

Miscellaneous Notes

- Like gpt-2-simple before it, the primary motivation behind releasing simpleaichat is to both democratize access to ChatGPT even more and also offer more transparency for non-engineers into how Chat AI-based apps work under the hood given the disproportionate amount of media misinformation about their capabilities. This is inspired by real-world experience from my work with BuzzFeed in the domain, where after spending a long time working with the popular LangChain, a more-simple implementation was both much easier to maintain and resulted in much better generations. I began focusing development on simpleaichat after reading a Hacker News thread filled with many similar complaints, indicating value for an easier-to-use interface for modern AI tricks.
 - simpleaichat very intentionally avoids coupling features with common use cases where possible (e.g. Tools) in order to avoid software lock-in due to the difficulty implementing anything not explicitly mentioned in the project’s documentation. The philosophy behind simpleaichat is to provide good demos, and let the user’s creativity and business needs take priority instead of having to fit a round peg into a square hole like with LangChain.
 - simpleaichat makes it easier to interface with Chat AIs, but it does not attempt to solve common technical and ethical problems inherent to large language models trained on the internet, including prompt injection and unintended plagiarism. The user should exercise good judgment when implementing simpleaichat. Use cases of simpleaichat which go against OpenAI’s usage policies (including jailbreaking) will not be endorsed.
 - simpleaichat intentionally does not use the “Agent” logical metaphor for tool workflows because it’s become an AI hype buzzword heavily divorced from its origins. If needed be, you can emulate the Agent workflow with a **while** loop without much additional code, plus with the additional benefit of much more flexibility such as debugging.
- The session manager implements some sensible security defaults, such as using UUIDs as session ids by default, storing authentication information in a way to minimize unintentional leak-

age, and type enforcement via Pydantic. Your end-user application should still be aware of potential security issues, however.

- Although OpenAI's documentation says that system prompts are less effective than a user prompt constructed in a similar manner, in my experience it still does perform better for maintaining rules/a persona.
- Many examples of popular prompts use more conversational prompts, while the example prompts here use more concise and imperative prompts. This aspect of prompt engineering is still evolving, but in my experience commands do better with ChatGPT and with greater token efficiency. That's also why simpleaichat allows users to specify system prompts (and explicitly highlights what the default use) instead of relying on historical best practices.
- Token counts for async is not supported as OpenAI doesn't return token counts when streaming responses. In general, there may be some desync in token counts and usage for various use cases; I'm working on categorizing them.
- Outside of the explicit examples, none of this README uses AI-generated text. The introduction code example is just a joke, but it was too good of a real-world use case!

Roadmap

- PaLM Chat (Bard) and Anthropic Claude support
- More fun/feature-filled CLI chat app based on Textual
- Simple example of using simpleaichat in a webapp
- Simple example of using simpleaichat in a stateless manner (e.g. AWS Lambda functions)

Maintainer/Creator

Max Woolf (@minimaxir)

Max's open-source projects are supported by his Patreon and GitHub Sponsors. If you found this project helpful, any monetary contributions to the Patreon are appreciated and will be put to good creative use.

License

MIT