**Python and OpenAI**

<https://platform.openai.com/docs/overview>

<https://realpython.com/practical-prompt-engineering/>

**Getting and Installing an OpenAI Key**

Python calls OpenAI through an API that requires an *API Key* for authentication.

(You can create API keys at a user or service account level. Service accounts are tied to a bot and should be used to provision access for production systems. Each API key can be scoped to (a) *project keys* or (b) *user keys*. Project keys provide access to a single project (preferred option). Access [*Project API keys*](https://platform.openai.com/settings/organization/api-keys) by selecting the specific project for which you wish to generate keys.

Organization IDs are found on your [Organization settings](https://platform.openai.com/settings/organization/general) page. Project IDs are found on your [General settings](https://platform.openai.com/settings/organization/projects) page.

*Steps*

1. Open an OpenAI developer account and buy some credits.

2. Click on the *API reference* link to gather some useful information:

A screenshot of a computer

Description automatically generated

3. Go to the OpenAI API keys page (link referenced above) and click on the *Create new secret key* button:

A screenshot of a computer

Description automatically generated

4. Complete the *Create new secret key* form:

A screenshot of a computer

Description automatically generated

[Note: I don’t know if I should have created a Project first, but I selected *Default project* on my first attempt.]

*MacOS Instructions for Installing the OpenAI Key*

1. On my computer, *.zshrc* and *.bash\_profile* are stored in my user home directory:

A black text with a white background

Description automatically generated

2. Use the *ls -a* command to see the hidden and unhidden files in this directory:

A screenshot of a computer code

Description automatically generated

3. My computer has both a *.zshrc* and *.bash\_profile* file, so I added the OpenAI API Key to both files. You can edit them using *nano* within the command terminal. For example, the use following command…



…to open the following window:

A screenshot of a computer

Description automatically generated

4. Add this line to each file:



Note: The API Key should be in quotation marks. Only a part of my key is showing, but there is a quotation mark at the end of the line. I put the key all one line. (It may be possible to wrap the key, but I *know* it works if you put it on one line.)

Note: The reason there are two configuration files is that there are two versions of Terminal – one called bash and the other called zsh. You can configure the Terminal shortcut to target either one. I happen to use zsh and I believe it is the newer of the two.

5. After you add “export OPENAI… [etc.]” to both the .zshrc and .bash\_profile files, re-start your terminal to activate the new configuration. (You do not need to restart the computer, only Terminal, which is what the .zshrc and .bash\_profile files configure.)

*Windows Instructions for Installing the OpenAI Key*

[to come – involves storing the key in the environment variables]

**Installing the OpenAI Package**

1. Create and activate venv (in this case, called *venv\_openai*):



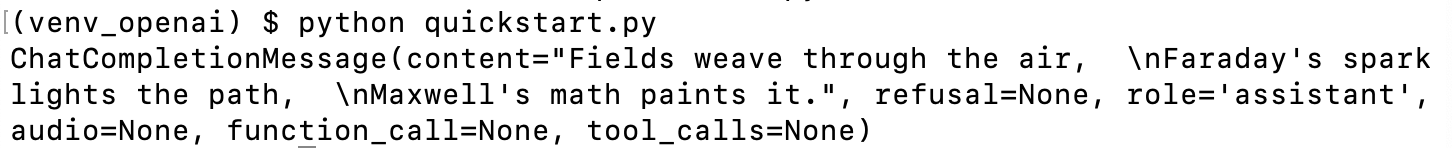
Note: The OpenAI version installed (by default) on my computer was 1.58.1

2. Create a file called quickstart.py and paste the following code into it:

A screenshot of a computer program

Description automatically generated

3. Save the file and run it in a venv as follows:



Note: Without any formatting, the message returned from OpenAI and printed will include line ending character and other metadata, but it writes a very fun, and even beautiful haiku:

Fields weave through the air,

Faraday’s spark lights the path,

Maxwell’s math paints it.

It’s not perfect, but not bad.

(continued on next page)

**Prompt Engineering**

<https://realpython.com/practical-prompt-engineering/>

The following are notes from a Real Python tutorial titled ***Prompt Engineering: A Practical Example***

1. Create a directory in a venv folder to hold several files that will be downloaded in the next step. (On one computer, I named the folder prompt\_eng\_materials.)

2. Download the project files from this location: <https://realpython.com/bonus/prompt-engineering-code/>

3. Use pip to install the required dependencies:

A close-up of a code

Description automatically generated

(On Windows, the command is: **python -m pip install -r requirements.txt** )

4. Run **pip list**, if you wish, to see the list of packages now installed:

A screenshot of a computer code

Description automatically generated

**Note: The instructions (here in these notes) assume that whether you have installed the API Key in your Terminal configuration files (on a Mac) or in the environment variables (on Windows), described at the opening of this document.**

5. Now run the app.py script with the following command:



Output:

A screenshot of a cell phone

Description automatically generated

6. In case Real Python takes down this tutorial, all of the following are the key files:

**A screenshot of a computer program

Description automatically generated**

A screenshot of a computer code

Description automatically generated

**Here is copyable text for app.py:**

import argparse

import os

import tomllib

from pathlib import Path

from openai import OpenAI

\_\_all\_\_ = ["get\_chat\_completion"]

# Authenticate

client = OpenAI(api\_key=os.getenv("OPENAI\_API\_KEY"))

# Load settings file

settings\_path = Path("settings.toml")

with settings\_path.open("rb") as settings\_file:

SETTINGS = tomllib.load(settings\_file)

def parse\_args() -> argparse.Namespace:

"""Parse command-line input."""

parser = argparse.ArgumentParser()

parser.add\_argument("file\_path", type=Path, help="Path to the input file")

return parser.parse\_args()

def main(args: argparse.Namespace) -> None:

file\_content = args.file\_path.read\_text("utf-8")

print(get\_chat\_completion(file\_content))

def get\_chat\_completion(content: str) -> str:

"""Send a request to the /chat/completions endpoint."""

response = client.chat.completions.create(

model=SETTINGS["general"]["model"],

messages=\_assemble\_chat\_messages(content),

temperature=SETTINGS["general"]["temperature"],

seed=12345, # Doesn't do anything for older models

)

return response.choices[0].message.content

def \_assemble\_chat\_messages(content: str) -> list[dict]:

"""Combine all messages into a well-formatted list of dicts."""

messages = [

{"role": "system", "content": SETTINGS["prompts"]["role\_prompt"]},

{"role": "user", "content": SETTINGS["prompts"]["negative\_example"]},

{

"role": "system",

"content": SETTINGS["prompts"]["negative\_reasoning"],

},

{

"role": "assistant",

"content": SETTINGS["prompts"]["negative\_output"],

},

{"role": "user", "content": SETTINGS["prompts"]["positive\_example"]},

{

"role": "system",

"content": SETTINGS["prompts"]["positive\_reasoning"],

},

{

"role": "assistant",

"content": SETTINGS["prompts"]["positive\_output"],

},

{"role": "user", "content": f">>>>>\n{content}\n<<<<<"},

{"role": "user", "content": SETTINGS["prompts"]["instruction\_prompt"]},

]

return messages

if \_\_name\_\_ == "\_\_main\_\_":

main(parse\_args())

**The settings.toml file that this script relies on has the following contents:**

[general]

chat\_models = ["gpt-3.5-turbo", "gpt-4"]

model = "gpt-3.5-turbo"

temperature = 0

[prompts]

instruction\_prompt = """

Remove personally identifiable information, only show the date,

and replace all swear words with "😤"

"""

role\_prompt = """

"""

positive\_example = """

"""

positive\_reasoning = """

"""

positive\_output = """

"""

negative\_example = """

"""

negative\_reasoning = """

"""

negative\_output = """

"""

**The chats.txt file contents are the following messages:**

[support\_tom] 2023-07-24T10:02:23+00:00 : What can I help you with?

[johndoe] 2023-07-24T10:03:15+00:00 : I CAN'T CONNECT TO MY BLASTED ACCOUNT

[support\_tom] 2023-07-24T10:03:30+00:00 : Are you sure it's not your caps lock?

[johndoe] 2023-07-24T10:04:03+00:00 : Blast! You're right!

[support\_amy] 2023-06-15T14:45:35+00:00 : Hello! How can I assist you today?

[greg\_stone] 2023-06-15T14:46:20+00:00 : I can't seem to find the download link for my purchased software.

[support\_amy] 2023-06-15T14:47:01+00:00 : No problem, Greg. Let me find that for you. Can you please provide your order number?

[greg\_stone] 2023-06-15T14:47:38+00:00 : It's 1245789. Thanks for helping me out!

[support\_louis] 2023-05-05T09:22:12+00:00 : Hi, how can I help you today?

[karen\_w] 2023-05-05T09:23:47+00:00 : MY BLASTED ORDER STILL HASN'T ARRIVED AND IT'S BEEN A WEEK!!!

[support\_louis] 2023-05-05T09:24:15+00:00 : I'm sorry to hear that, Karen. Let's look into this issue.

[support\_louis] 2023-05-05T09:25:35+00:00: Can you please provide your order number so I can check the status for you?

[karen\_w] 2023-05-05T09:26:12+00:00: Fine, it's 9876543.

[support\_louis] 2023-05-05T09:26:45+00:00: Thank you, Karen. I see there was a delay in shipping. Your order will arrive within the next 2 days.

[support\_jenny] 2023-06-18T17:35:28+00:00: Hello! How can I help you today?

[alex\_harper] 2023-06-18T17:36:05+00:00: I accidentally placed an order twice, can you help me cancel one?

[support\_jenny] 2023-06-18T17:36:25+00:00: Sure, Alex. Can you give me the order number you'd like to cancel?

[alex\_harper] 2023-06-18T17:36:55+00:00: Yes, it's 1122334. Thank you!

[support\_jenny] 2023-06-18T17:37:32+00:00: I've successfully canceled order number 1122334. You will receive a confirmation email shortly.

[support\_ben] 2023-06-29T11:51:45+00:00: Good morning, what can I assist you with today?

[lisa\_beck] 2023-06-29T11:52:20+00:00: Hi there, I received a damaged item in my order. Can you help me return it?

[support\_ben] 2023-06-29T11:52:45+00:00: I'm sorry to hear that, Lisa. Can you provide your order number and specify the damaged item?

[lisa\_beck] 2023-06-29T11:53:22+00:00: Sure, order number is 5566778 and the damaged item is a coffee mug.

[support\_rachel] 2023-05-04T08:16:37+00:00: How can I help you today?

[mike\_t] 2023-05-04T08:17:15+00:00: My coupon code isn't working at checkout. Can you help?

[support\_rachel] 2023-05-04T08:17:38+00:00: Of course, Mike. Please provide the coupon code you're trying to use.

[mike\_t] 2023-05-04T08:18:02+00:00: It's "HELLO10".

[support\_rachel] 2023-05-04T08:18:37+00:00: I've checked the code, and it seems to have expired. I apologize for the inconvenience. Here's a new code for you to use: "WELCOME15".

[support\_vincent] 2023-06-15T20:43:55+00:00: Good evening! How may I assist you?

[sara\_winters] 2023-06-15T20:44:30+00:00: Hi there, I'm having trouble logging into my account. I've tried resetting my password, but it's not working.

[support\_vincent] 2023-06-15T20:44:52+00:00: I'm sorry to hear that, Sara. Let me help you. Can you please confirm your email address?

[sara\_winters] 2023-06-15T20:45:25+00:00: Sure, it's sara.winters@email.com.

[support\_david] 2023-06-24T16:28:43+00:00: Welcome! What can I do for you today?

[jane\_d] 2023-06-24T16:29:16+00:00: Hi, I need to change my delivery address for my recent order.

[support\_david] 2023-06-24T16:29:43+00:00: Alright, Jane. Please provide your order number.

[jane\_d] 2023-06-24T16:30:11+00:00: It's 3344556. Thanks for your help!

**Notes on Prompt Engineering Code**

The Python script defines a **get\_chat\_completion()** function, which in turn is a wrapper for the **client.chat.completions.create()** OpenAI function, which uses the OpenAI **/chat/completions** endpoint. The full call to the OpenAI function with parameters instantiates and object named *response* (shown below in color to make it easier to see the parameters). A method of the *response* object (.choices[].message.content, which in this case is the *edited* version of the texts sent to OpenAI.

response = client.chat.completions.create(

model=SETTINGS["general"]["model"],

messages=\_assemble\_chat\_messages(content),

temperature=SETTINGS["general"]["temperature"],

seed=12345, # Doesn't do anything for older models

)

A method of the *response* object (.choices[].message.content, which in this case is the *edited* version of the texts sent to OpenAI:

return response.choices[0].message.content

The /chat/completions create() function uses (in this script) the following parameters:

* **model** (mandatory) – ID of the model use (see the [model endpoint compatibility](https://platform.openai.com/docs/models#model-endpoint-compatibility) table for details on which models work with Chat API). \*\*
* **messages** (mandatory) – list of messages to be worked with. (Another function in the Python script (**\_assemble\_chat\_messages()** ) prepackages the messages.
* **temperature** (not mandatory) – sets how “deterministic” (i.e., predictable) the AI’s response will be. (This is pulled from the .toml configuration file and held in the SETTINGS dictionary [?].
* **seed** (not mandatory) – a feature that is currently in beta (@2025/1/1). If specified, OpenAI will make a best effort to sample deterministically, so that repeated requests with the same *seed* and parameters should return the same result.

There are many more parameters that can be set.

**How File Paths Are Set**

print(get\_chat\_completion(file\_content)) [line 27]

Note: **file\_content** is set with the **args.file\_path.read\_text("utf-8")** function.

In turn, **args.file\_path…** relies on the argparse library imported at the top of the script [line 1]. *Argparse* is a library that gives access to information about the system environment, in this case, the file path to the text data file. (This makes the script run in any directory on the computer, as long as the external files are in the same location as the script.)

**Temperature Setting**

The level of consistency in the response from OpenAI is set by the *temperature* setting, which is found in the settings.toml file:

A screenshot of a computer

AI-generated content may be incorrect.

When you integrate an LLM into a product or workflow, you’ll often want deterministic responses. Note that you currently can never get 100% consistency. Inference (predicting the next token based on the ones that precede it) is non-deterministic even at temperature = 0 when the top 2 token probabilities are less than 1% difference.

It should also be noted that OpenAI is continuously upgrading their models, so over time, you will interact with different LLMs and get different results.

**Start Engineering Your Prompts**

The following are techniques for getting the desired output from an LLM:

* **Zero-shot prompting**: Giving the language model normal instructions without any additional context
* **Few-shot prompting**: Conditioning the model on a few examples to boost its performance
* **Using delimiters**: Adding special tokens or phrases to provide structure and instructions to the model
* **Detailed, numbered steps**: Breaking down a complex prompt into a series of specific steps

An example of zero-shot prompting (the “zero” refers to the number of examples provided) is following instruction:

A close-up of a white background

AI-generated content may be incorrect.

When sending the chat records (shown above in chats.txt), you get the following response:

A screenshot of a computer code

AI-generated content may be incorrect.

You’ll notice that there’s an angry emoji for the 2023-06-15 chat, even there is no profanity in the chat. This is true of for several other days’ chats as well. In short, it didn’t do a poor job of responding to the instructions. This is a good example of why zero-shot prompting isn’t very useful. The LLM understood that it should do something with the huffing emoji and reduce the ISO data-time to only a date, but failed to make other distinctions, so the results have no value.

**Few-shot Prompting**

In the following code, the AI is now given an example (outlined in red):

A screenshot of a computer

AI-generated content may be incorrect.

(Note: I also add the instructions outline in green because at first, when I only gave the AI one example, it only returned a response for the one chat that matched the example.)

(continued on next page)

Now we get the following:

A screenshot of a computer message

AI-generated content may be incorrect.

It did a much better job. However…

It replaced all of the names at the beginning of each line, but it included the personal names that appear in the message, such as “Greg” and “Karen.” It also didn’t hide the personal information of order numbers and email addresses.

(continued on next page)

**Adding More Examples**

Adding a second example helps. (However, in this case it also causes OpenAI to get more stubborn about only replying with the examples.) See the following for an example of this behavior:

(sample code, giving AI two examples)

A screenshot of a computer

AI-generated content may be incorrect.

(Output Response)

A screenshot of a chat

AI-generated content may be incorrect.

**Using Delimiters to Improve the Results**

There are times when using delimiters such #### or ------ to clearly mark segments of meaning will help, as in the following:

A screenshot of a computer error

AI-generated content may be incorrect.

Adding section delimiters can be a good practice, but it doesn’t solve the problem of it only returning the two cases which match the examples. This may be a case of “overfitting.” It’s a bad practice to use training data (the examples) that is the same as the data you’re trying to work with. So, keeping the code the same, let’s run the same script against another batch of chat messages. The output with the new chat log (testing-chats.txt) are as follows:

(continued on next page)

A screenshot of a computer

AI-generated content may be incorrect.

Note: Now it’s almost a complete failure. The only thing it successfully does is reformat the dates. This means…

The prompt is poorly engineered.

**Describe Your Request in Number Steps**

We’ll now replace the instruction prompt with clearly numbered and delineated steps:

A computer error message

AI-generated content may be incorrect.

Now we get the following results:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The results are obviously better. Giving numbered, step-by-step instructions gives you a much more consistent outcome.

(continued on next page)

*Glossary*:

**model endpoint compatibility** – A *model* is a reasoning model, meaning that it’s an LLM that has been adapted or trained for various general or specific purposes. Below is a current table (as of 2024/01/01) of the current reasoning models available. The script above uses the /v1/chat/completions model.

A screenshot of a computer

Description automatically generated

**Endpoint** – An API interface. In this case of the script above, the API endpoint is <https://api.openai.com/v1/chat/completions>)

**o1 reasoning models** – You must be a “Tier 5” customer (paid $1,000 with a 30+ day wait) to use this model. OpenAI o1 series reasoning models are new large language models trained with reinforcement learning to perform complex reasoning. o1 models “think before they answer.” o1 models excel in scientific reasoning, currently ranking in the 89th percentile on Codeforces (a competitive programming evaluation tool. o1 models placed among the top 500 students in the US in a qualifier for the USA Math Olympiad (AIME), and exceeding human PhD-level accuracy on benchmark physics, biology, and chemistry problems (GPQA).

There are two reasoning models available in the API:

* o1 – designed to reason about hard problems using broad general knowledge about the world
* o1-mini – a faster and more affordable version of o1, particularly adept at coding, math, and science tasks where extensive general knowledge isn’t required.

**parameter support** – [to come]