

Introduction to Developing Generative AI Web Apps Class: Lessons 4-7

[Lesson 4: The Chat Completions API Endpoint](#)

- [4.1 What is the Chat Completions API?](#)
- [4.2 Anatomy of a Chat Completions API Call](#)
- [4.3 Anatomy of a Chat Completions API response](#)
- [4.5 Write your first API call!](#)

[Lesson 5: Working With Message Roles](#)

- [5.1 Understanding Message Roles](#)
- [5.2 Review the Three Message Roles](#)
- [5.3 Give your chatbot some personality with the system role](#)
- [5.4 Practice Writing Prompts with the user role](#)
- [5.5 Single-turn vs. multi-turn conversations: what's the difference?](#)
- [5.6 Refactor your chatbot to be multi-turn with the assistant role](#)

[Lesson 6: Tokens](#)

- [6.1 What are tokens?](#)
- [6.2 Evaluate Your Chatbot's Token Usage](#)
- [6.3 Tactics for Managing Token Usage](#)
- [6.4 Limit your chatbot's token usage](#)
- [6.5 Log your chatbot's token usage](#)

[Lesson 7: The Assistants API](#)

- [7.1 What is the Assistants API?](#)
- [7.2 How assistants work: anatomy of an assistant](#)
- [7.3 How Assistants Work: Breakdown of the 4 Objects](#)
- [7.4 How to build an assistant](#)
- [7.5 Start building your assistant](#)
- [7.6 Zhuzh up your study buddy assistant](#)

Lesson 4: The Chat Completions API Endpoint

4.1 What is the Chat Completions API?

In this lesson, you'll learn:

- What the Chat Completions API is
- The anatomy of an API call
- The anatomy of a Chat Completions API response
- And you'll practice writing your very own API call

Let's begin!

In the last lesson you set up your OpenAI API account and made sure your environment is all ready to go, and NOW it's time to crack open the hood and really take a look at how the OpenAI API works for their GPT model.

As we've mentioned, in this class you will be using the GPT 3.5 turbo model.

And to begin, we will use the Chat Completions API to create our first, very simple chatbot.

The Chat Completions API is one of two APIs we can use to build our chatbot, the other option being the Assistants API. Ultimately, we will be switching to the Assistants API because it provides a more robust set of features that we'll want our web application to be able to use.

But for now, the Chat Completions API is perfect for us because it's a simpler API endpoint that is easy to get up and running. In real-world applications, the Chat Completions API is best for single chatbot interactions, like customer support, where you don't necessarily need to keep a record of all previous interactions.

You will use the Chat Completions API exactly as you've used APIs in the past—you'll send a request to an endpoint, the endpoint will receive the request, process the request, and return a response to you.

The only difference here is that when you use the Chat Completions API, it'll process the request using OpenAI's GPT large language model.

So let's get started!

4.2 Anatomy of a Chat Completions API Call

Now that you've learned what the Chat Completions API endpoint is and what it can be used for, let's dive into the technical details behind an API call.

CONNECTING TO THE API

The first thing you must do before you can use the OpenAI API—or any API for that matter—is to connect to the API and give it your API key.

This is done by first importing the OpenAI library you installed in a previous lesson into your Python file:

```
from openai import OpenAI
```

And then creating an instance of the OpenAI API in order to interact with the endpoints:

```
client = OpenAI()
```

And that's it! You'll notice that you're not entering your API key explicitly—this is by design. `OpenAI()` by default will look for a system variable called `OPENAI_API_KEY` with a value assigned to it. As long as those two things exist and `OPENAI_API_KEY` is assigned a valid API key, it works automagically!

THE REQUEST BODY

A Chat Completions API call is formatted as an object, known as the request body, and has two required parameters: model and messages.

Model

The model parameter is a string that represents the ID of the model to use for the request.

When you start reviewing extra resources and exploring the OpenAI documentation, you'll notice that OpenAI has several different models that can be used with the Chat Completions API endpoint. These models have slightly different capabilities and configurations, and it can be difficult to understand which model to use. Look at the current list of GPT chat models for example:

```
/v1/chat/completions    gpt-4 and dated model releases, gpt-4-turbo-preview and dated  
                        model releases, gpt-4-vision-preview, gpt-4-32k and dated  
                        model releases, gpt-3.5-turbo and dated model releases, gpt-3.5-  
                        turbo-16k and dated model releases, fine-tuned versions of gpt-  
                        3.5-turbo
```

As of March 2024.

How does a budding AI engineer know which one to choose?? The big choice to make is choosing between the gpt-3.5-turbo and gpt-4 models—they both have pros and cons and choosing between the two depends on your specific use case.

In this class, we'll be using **gpt-3.5-turbo** because it's much more economical than gpt-4. The main advantages of gpt-4 are its speed and the fact that it returns more contextually relevant results. However, gpt-3.5-turbo is a fine choice for educational purposes and even smaller web applications!

Messages

The messages parameter is an array of objects, with each object containing two required fields: role and content. Role defines the author of the content, and content defines the content being passed to the model. This content is the prompt that the model uses to generate text output.

Don't worry about the details for now, you'll learn much more about this in the next lesson!

A simple request using the two required parameters looks like this:

```
completion = client.chat.completions.create(  
    model="gpt-3.5-turbo",  
    messages=[  
        {"role": "system", "content": "You are a helpful assistant."},  
        {"role": "user", "content": "Hello!"}  
    ]  
)
```

Optional Parameters

In addition to the required model and messages parameters, there are several other optional parameters that can be passed to the endpoint. And you'll work with a few of them in a future lesson. You can learn more about *all* of the parameters in the [OpenAI API documentation](#).

4.3 Anatomy of a Chat Completions API Response

Now that you've taken a look at a Chat Completions API request, let's see what the API's response looks like.

After the model processes the input (aka the request), it returns data in the form of an object. This object contains unique identifying information, the output generated by the model, and some other useful information. Let's dig in!

In the last step, we reviewed an API request that looked like this:

```
completion = client.chat.completions.create(  
    model="gpt-3.5-turbo",  
    messages=[  
        {"role": "system", "content": "You are a helpful assistant."},  
        {"role": "user", "content": "Hello!"}  
    ]  
)
```

If you look at the `completion` object using Python's `print()` command, this is what you'll see (this response contains example data):

```
{  
    "id": "chatcmpl-123",  
    "object": "chat.completion",  
    "created": 1677652288,  
    "model": "gpt-3.5-turbo-0125",  
    "system_fingerprint": "fp_44709d6fcb",  
    "choices": [{  
        "index": 0,  
        "message": {  
            "role": "assistant",
```

```

        "content": "\n\nHello there, how may I assist you today?",
    },
    "logprobs": null,
    "finish_reason": "stop"
  ],
  "usage": {
    "prompt_tokens": 9,
    "completion_tokens": 12,
    "total_tokens": 21
  }
}

```

Let's review each of these properties and their types, values, and how to reference them using Python. The properties you'll be using in future lessons are starred ★.

If you need a refresher on data types in Python, refer back to the [Building Variables & Data Types](#) lesson.

id (string)

The unique identifier for the chat completion object.

`completion.id`

object (string)

The object type, which is always `chat.completion`. This is handy for when you have multiple requests going to different types of AI models and need to make sure you're working with the right response objects in your code.

`completion.object`

★ **choices** (array)

A list of choices, which are simply the conversational responses generated by the model from the prompt in the API request. By default the model will return one choice, but you can specify how many possible responses you want by using the `n` parameter in the request.

`completion.choices` will return all choices in the array

`completion.choices[0]` will return the first choice in the array

`completion.choices[0].message.content` will return the text generated by the model in response to the prompt in the API request

This property is one you will be using a lot in this class!

created (integer)

The [Unix](#) timestamp of when the response was created. This looks like a long string of numbers and it refers to the number of seconds that have elapsed since 1/1/1970 (neat, huh?). Unix timestamps are a way for developers all over the world to refer to the exact same time regardless of location.

`completion.created`

model (string)

The model used for the response. In almost *all* cases this will be the model you specified in your request, but it will also include the exact version number. In the example above, you'll see the exact version of the gpt-3.5-turbo model used is 0125.

`completion.model`

system_fingerprint (string)

The system fingerprint represents the backend configuration that the model runs with. You can think of this as sort of like a specific version of an app running on a specific version of an operating system, like the latest version of the Instagram app running on the latest version of iOS. The app needs to know what version of iOS it's running on in order to work correctly for that specific version. This system_fingerprint is a record of that kind of relationship for the API request.

`completion.system_fingerprint`

★ **usage** (object)

Token usage information for both the API request and the API response. You'll learn more about tokens in a future lesson!

`completion.usage.prompt_tokens` will return the total number of tokens used by the API request

`completion.usage.completion_tokens` will return the total number of tokens used by the API response

`completion.usage.total_tokens` will return the sum of the two

4.5 Write your first API call!

Now it's time to bring together all your new knowledge about the chat completion API endpoint! In this challenge you're going to create a Python script that:

1. Accepts an input from the user
2. Takes that input and makes a request to the Chat Completions API endpoint
3. Extracts the response message content and prints it out to the user

It's almost like... you'll be having a brief conversation with AI!

Since this is your first code-based challenge in this class, we'll be going line by line so that you can take your time and practice.

CREATE YOUR FILE AND ACCEPT AN INPUT

1. First we need to create the Python file we'll be using for the next several challenges. In the command line, navigate to your chatbot project directory.
2. We'll be working in the virtual environment. If your virtual environment is not currently running (you don't see (openai-env) before your prompt) start it by running one of the following commands:

On a Mac:

```
source openai-env/bin/activate
```

On Windows:

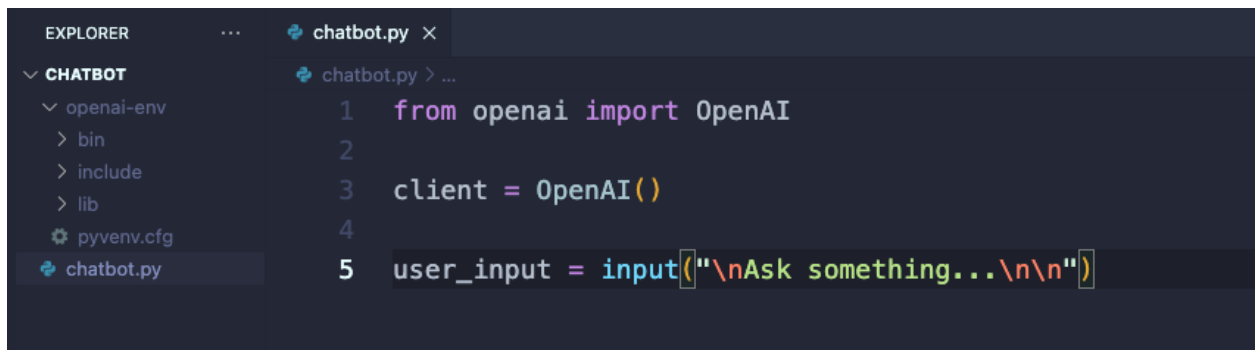
```
openai-env\Scripts\activate
```

3. Next, create a new file called chatbot.py with the `touch` command. Open up the chatbot directory in your text editor.
4. The first thing we need to do is import the OpenAI library. At the top of the chatbot.py file add: `from openai import OpenAI`
5. Importing the library isn't useful by itself. In order to interact with the API, we need to create an instance of the API library and store it in a variable, which is done by adding this line of code to the chatbot.py file: `client = OpenAI()`

Remember, you don't need to enter your API key explicitly because `OpenAI()` will pull the value from your `OPENAI_API_KEY` system variable.

- Next, we need to ask our user to submit text. Since we'll be running this Python script via the command line—at least to start—we need to prompt the user to enter something with the `input()` function. Add the next line of code to the `chatbot.py` file:
`user_input = input("\nAsk something...\n\n")`

- At this point, your `chatbot.py` file will look like this:



```
EXPLORER  ...  chatbot.py x
  CHATBOT
    openai-env
      > bin
      > include
      > lib
      pyvenv.cfg
      chatbot.py
  chatbot.py > ...
1  from openai import OpenAI
2
3  client = OpenAI()
4
5  user_input = input("\nAsk something...\n\n")
```

- To really understand how this works, save the file, head over to the command line, and run your Python script:

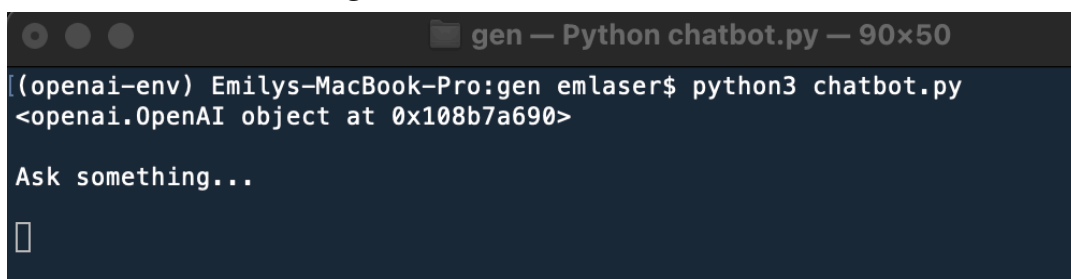
On a Mac:

```
python3 chatbot.py
```

On Windows:

```
python chatbot.py
```

- You should see something that looks like this:



```
gen — Python chatbot.py — 90x50
[(openai-env) Emilys-MacBook-Pro:gen emlaser$ python3 chatbot.py
<openai.OpenAI object at 0x108b7a690>

Ask something...

]
```

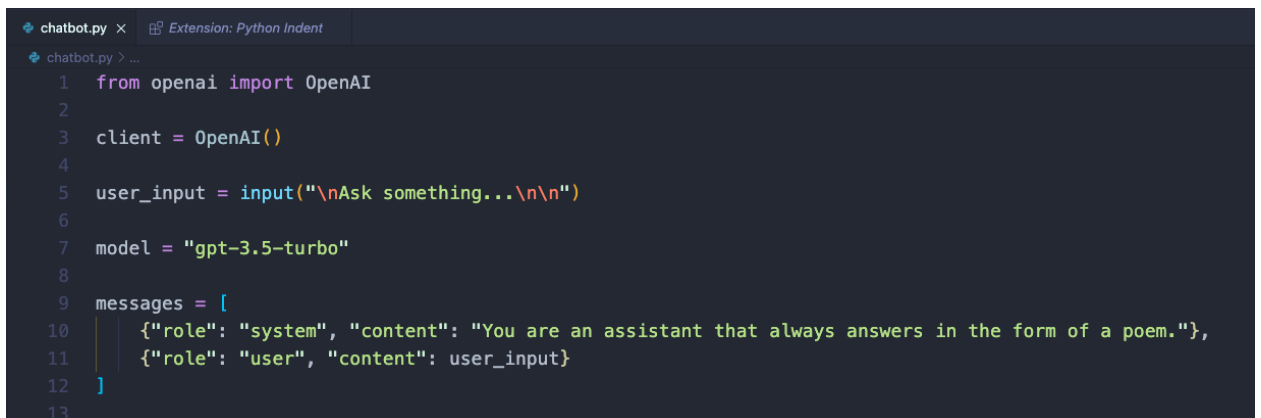
10. You can see that we have a Python script waiting for the user to submit text, like a question or statement. Go ahead and type anything here and hit enter. You won't see anything happen since we haven't yet written the code to process the user's text, but this will stop the script from running. Behind the scenes, what you typed is stored in the `user_input` variable.
11. You can comment out or remove this line from your code.

SEND A REQUEST TO THE CHAT COMPLETIONS API ENDPOINT

Now that we have the user's submitted text stored in a variable, we can use that to send a request to OpenAI's chat completions API.

1. For this request, we'll use our friend the `chat.completions.create()` function. As you learned in a previous lesson, there are only two required parameters for this function: model and messages. Let's define them!
2. For model, we'll add a string value representing the model we want to use: `model = "gpt-3.5-turbo"`
3. And for messages, we need to create an array of message objects for the system and user roles:

```
messages = [  
    {"role": "system", "content": "You are an assistant that always answers in the form of a poem."},  
    {"role": "user", "content": user_input}  
]
```
4. The code you've written so far should look like this:



```
chatbot.py x  Extension: Python Indent  
chatbot.py > ...  
1  from openai import OpenAI  
2  
3  client = OpenAI()  
4  
5  user_input = input("\nAsk something...\n\n")  
6  
7  model = "gpt-3.5-turbo"  
8  
9  messages = [  
10     {"role": "system", "content": "You are an assistant that always answers in the form of a poem."},  
11     {"role": "user", "content": user_input}  
12 ]  
13
```

5. As you can see, we're defining the behavior or personality of the model's response using the system role, and we're sending the user's submitted text using the user role.

- Now we have what we need to build our API request, which we will assign to a variable so that we can store the response. Below the messages array, add:
`response = client.chat.completions.create(
 model = model,
 messages = messages
)`
- Ooo now we're cookin'. Print the response so we can run our script and see the AI's response. Add `print(response)` below the `response = chat.completions.create()` function.
- Save your file and run the script. You should see something like this:

```
chatbot — -bash — 90x50
(openai-env) Emilys-MacBook-Pro:chatbot emlaser$ python3 chatbot.py
Ask something...

what is the cutest animal?
ChatCompletion(id='chatcmpl-94J6JYZjnEBcQ5H54IY5wn1YVoD2u', choices=[Choice(finish_reason='stop', index=0, logprobs=None, message=ChatCompletionMessage(content="In the forest or down by the sea,\nThere's an animal that's as cute as can be.\nWith fluffy fur and eyes so bright,\nIt's the otter that steals the limelight.\n\nSliding playfully on its belly,\nChasing fish, feeling quite smelly.\nWith whiskers twitching, always on the go,\nThe otter's cuteness really does show.\n\nSo if you're looking for an animal so sweet,\nThe otter is the one that can't be beat.\nWith its playful nature and joyful glee,\nThe otter is the cutest, don't you agree?", role='assistant', function_call=None, tool_calls=None))], created=1710813763, model='gpt-3.5-turbo-0125', object='chat.completion', system_fingerprint='fp_4f2ebda25a', usage=CompletionUsage(completion_tokens=123, prompt_tokens=32, total_tokens=155))
(openai-env) Emilys-MacBook-Pro:chatbot emlaser$
```

It's a bit hard to read, but there is our response! Woohoo!! Let's dig in.

EXTRACT THE MESSAGE CONTENT AND PRINT IT OUT TO THE USER

If you've used ChatGPT, or any kind of chatbot for that matter, you're probably quite familiar with the fact that this is not how a chatbot typically responds. A chatbot's response feels more like a conversation, right? This response has a whole lot more information in it, which you learned all about in a previous lesson.

So, what we need to do is extract *just* the message from this response so we can print that out to the user.

- Directly above your `print(response)` line, add a new line of code that extracts the message content and assigns it to a new variable: `response_for_user = response.choices[0].message.content`

2. Update your `print(response)` line to instead print your new variable: `print(response_for_user)`. Save the file and run your script.

(!) What is this “\n” thing I keep seeing?

Simply put, this is called an "escape sequence" and it creates a new line within a string. This is a handy way of making inputs and outputs more readable when interacting with scripts in the command line. Go ahead and try it out! Update your `print(response_for_user)` line to `print("\n" + response_for_user + "\n")`, save, and run your script again. See the difference??

And there you have it! You just created a very simple chatbot. PRETTY COOL, huh?? Yeah it is!

You have one final thing to do. Now that you have working code, you need to create a Git repository and push your code to your GitHub account. Enter the link to your new GitHub repo below and mark this challenge as complete! Oh, and if you're all done for now, don't forget to deactivate your virtual Python environment by running the `deactivate` command.

Take a moment to compare your code to [the solution code](#).

4.6 Refactor the chatbot script

In the last challenge, you created a simple Python script that serves as a chatbot. While your code is working just fine, it could be better!

Your challenge is to refactor your code so that the majority of the logic is inside a function. To keep from overwriting your working code from the last lesson, in the command line create a new branch for this lesson.

We aren't going to guide you through this challenge because we know you can do it! That being said, don't expect to get this right the first time. This is meant to challenge you to integrate some of the things you've learned in previous Python classes.

We WILL give you a few hints to get you started:

- Open up the Python script from the last challenge. Since you created a branch for this lesson, you should have an exact copy of the code and you can make changes while preserving the code from the previous lesson.
- Start by creating a function using `def()` and then deciding what code you can move into the function. Don't forget to call your function!
- As you work on refactoring your code, run the script in the command line. If you've made any errors, you'll see error messages that will help you debug.

Here are some resources and suggestions to help you if you get stuck:

- Copy and paste any error messages into Google.
- Or better yet, use ChatGPT and ask it to explain the error message.

And don't forget, if you need a refresher on defining functions in Python, you can always refer to the [Fun With Functions lesson](#) in the first Python class.

Good luck! When you're done, be sure to add and commit your changes and push this branch to your GitHub repository. (Don't merge this branch into the main branch. You'll need the code on the main branch in a future lesson.) Submit your GitHub repo link below and mark this challenge as complete!

You can compare your code with [the solution code here](#).

Lesson 5: Working With Message Roles

5.1 Understanding Message Roles

In this lesson you'll learn:

- What the three different roles are in the OpenAI API and what they each do
- How prompts operate in the API and why they're much more powerful than the prompts you've been writing to ChatGPT
- You'll start to get familiar with how prompt engineering works and look at lots of examples of prompt engineering in action
- What a "turn" is in this context and build more robust chat experiences with multi-turn conversations
- And you'll begin the process of writing the code for your very own chatbot!

Let's begin with understanding Message Roles.

When you're writing calls to the Chat Completions API you will be doing so via 3 different "message roles," for example as you can see in this code I have three sections that each send directions to three different roles: the system, the user, and the assistant.

Let's take a moment to break down what each one of these roles does:

- System role: the system role is the role that holds all the general knowledge about the context of this application and any specific settings that you put in place. The system role governs the behavior of the assistant role. The system role also keeps track of the conversation and can even keep track of previous conversations the user has had.
- User role: the user role is exactly what it sounds like—the data input by the user in the form of the prompts they provide to your AI application.
- Assistant role: the assistant role is the role that provides the AI model's response to the user's prompt, but the assistant will adjust the response depending on the directions given to it by the system role. The assistant can also keep track of its own prior responses.

From a high level, what you, the developer will be doing with these roles is first priming the model with the expected tone and context (acting as the system role), then using the user role and the assistant roles to prime the model for what sort of input it should expect and what sort of output you would like it to provide.

Now, let's look at each one individually in the context of a simple chatbot.

In this code example, we have written the instruction to the model to be a "friendly and helpful assistant" in the the system role, and then primed it with a user and assistant back and forth and demonstrates the types questions the model can expect from their users, and examples of how we'd like them to respond.

Of course, in this instance all the input and output is hardcoded and the tone, context, and inputted data are very simple, but that won't always be the case! But before we get into all that, let's try playing with message roles!

5.2 Review the Three Message Roles

Before you get started on the challenges in this lesson—you have three of them!—let's review the message roles in the Chat Completions API endpoint:

1. **System role:** The system role holds all the general knowledge about the context of this application and any specific settings that you put in place. It's where you tell the AI who they are, i.e. "You are a friendly and helpful assistant." And it governs the behavior of the assistant role. The system role also keeps track of the conversation and can even keep track of previous conversations the user has had.
2. **User role:** The user role is exactly what it sounds like—the data input, or prompts, the user provides to your AI application.
3. **Assistant role:** The assistant role provides the AI model's response to the user's prompt. The assistant will adjust the response depending on the directions given to it by the system role. The assistant can also keep track of its own prior responses.

This code example shows all three message roles in use:

```
completion = client.chat.completions.create(  
    model = "gpt-3.5-turbo",  
    messages=[  
        {"role": "system", "content": "You're a friendly and helpful assistant."},  
        {"role": "user", "content": "Tell me a joke."},  
        {"role": "assistant", "content": "Sure! Here's an example of a joke: Why don't  
scientists trust atoms? Because they make up everything!"},  
        {"role": "user", "content": "That's funny! Can you tell me another one?"},  
        {"role": "assistant", "content": "Of course! Here's another joke: Why did the  
math book look sad? Because it had too many problems."}  
    ]  
)
```

You'll focus on one role in each of the three challenges in the rest of this lesson, and when you're done, you'll have a more user-friendly and conversational chatbot. Let's get to it!

5.3 Give Your Chatbot Some Personality With the System Role

Now that you've deepened your understanding of message roles, it's time to put this knowledge to use!

As a reminder, there are three different roles in the chat completions API endpoint: system, user, and assistant. In this challenge you'll learn more about the system role by:

- Changing the system role content to see how the model responds
- Creating a function that categorizes user input and adjusts the output accordingly

EXPERIMENT WITH THE SYSTEM ROLE CONTENT

You'll build off of the previous lesson's code so start by creating a branch of the code from this lesson. (If you haven't changed branches, you should still be on the branch from Lesson 4.)

1. Open up your chatbot.py file. Locate the variable that defines the message roles (yours may look slightly different):

```
messages = [  
    {"role": "system", "content": "You are an assistant that always answers in the form  
of a poem."},  
    {"role": "user", "content": user_input}  
]
```

2. Update the content of the system role to something else, anything you want! Here are some suggestions:
 - a. You are an assistant that loves literature.
 - b. You are an assistant that talks like a Shakespearian pirate.
 - c. You are an assistant that answers as if you're a detective solving a mystery.
3. Now save your file and run your script. How does the model respond based on changes to your system role? Pretty neat, huh?? 😊

The examples above are quite simple, but you can get REALLY creative! Check out [this GitHub repo](#) for some fun and engaging examples of prompts that can be used as system messages, and try some of them out.

CREATE A FUNCTION TO CUSTOMIZE THE OUTPUT

Next you're going to write a function that accepts the user content as a parameter and attempts to define what kind of input that content is.

Notice that I said “*attempts* to define”. It’s important to recognize that we have no control over how a user types their message. For example, a user could ask a question and not include a question mark at the end. We can take steps to identify this as a question, but there’s really no way to be 100% sure beyond a shadow of a doubt what the user’s intent is.

It should also be noted that this isn’t necessarily something you would do in a large generative AI application. This is more meant to show you the flexibility and capabilities of the system role within the chat completions API endpoint.

1. Toward the top of your chatbot.py file, above the function you created in the last step, create a new function called `set_user_input_category()` that accepts one parameter, the user’s input:

```
def set_user_input_category(user_input):
```

2. Within your new function, you need to check `user_input` to determine if it has any text within it that hints that it’s a question.
3. In order to do this, you need to create a dictionary called `question_keywords` that contains words and symbols often seen in questions. Let’s call these keywords. You can use the [The Five W's and H](#) (and a question mark to boot!):

```
def set_user_input_category(user_input):  
    question_keywords = ["who", "what", "when", "where", "why", "how", "?"]
```

4. Next you need to search `user_input` for each keyword within your dictionary to determine if you have a match, meaning, you need to see if the user’s submitted text contains who, what, when, where, why, how, or a question mark. Another way of saying this is that you need to search the `user_input` string for a substring of one of those keywords.
5. If the text *does* contain one of those words (or a question mark), you can reasonably assume that the user asked a question. If not, you can consider the user’s submitted text a statement.

6. To achieve this, you need to use a `for` loop to search `user_input` for each keyword one by one:

```
def set_user_input_category(user_input):  
    question_keywords = ["who", "what", "when", "where", "why", "how", "?"]  
    for keyword in question_keywords:  
        if keyword in user_input.lower():  
            return "question"  
    return "statement"
```

7. Let's break this down. The `for` loop is looping through `question_keywords` and searching the user's submitted text for each keyword, one at a time. If the keyword is found within `user_input`, the function returns "question". After the `for` loop has looped through every keyword in the `user_input`, the function returns "statement".
8. Finally, the `.lower()` method is used to ensure that `user_input` is all lowercase when it's searched for a lowercase keyword (aka substring). This is because the lowercase and uppercase versions of letters are considered different characters and will *not* match when compared to each other! (an "a" is not an "A"!)
9. Your function is done! What are you actually DOING with this function, you ask? You can use this function to tailor the output you deliver back to the user!
10. Locate the `response_for_user` variable, which should contain the extracted message from the chat completions API call. Below this line of code, call your function and pass in the user's submitted text. Check to see if the returned value from the function is "question", and if it is, tailor the output message by concatenating "Good question! " to the beginning of it:

```
response_for_user = get_api_chat_response_message(model, messages)  
  
if set_user_input_category(user_input) == "question":  
    response_for_user = "Good question! " + response_for_user
```

11. Save your file and run your script a few times, testing it with different types of questions and statements. If you need a little more guidance, try each of the following:

- a. What time is it
- b. Tell me a joke
- c. I love bananas
- d. Do you love bananas?

If your script is working correctly, the output from #1 and #4 should include “Good question!”, and #2 and #3 should not. Did it work??

(!) When you're running your script repeatedly in the command line, it can get a little tedious to type it in each time, right? Try using the up arrow key on your keyboard. You'll be able to scroll through the commands you've run. So much easier! 😊

When you're finished, don't forget to add and commit your updated script and push the commits on this branch to your GitHub repo.

You can have a look over [the solution code here](#). It's a great way to check the indentation of your code. This is Python after all!

5.4 Practice Writing Prompts With the User Role

When developers use tools like OpenAI to build large applications, one of the big goals is to write system prompts that are optimized to meet users' needs. In order to achieve this, it's crucial that YOU, future developer, understand the needs of your app's users and what kinds of questions they'll be asking by getting into their shoes. So, in this challenge, you'll take the role of the user and write several prompts.

However, because you haven't yet built the chatbot user interface, you'll still be writing these prompts directly in your script and running it in the command line to see the output. Don't worry, though — by the end of class you'll have a fully functional chatbot complete with a user interface!

Imagine you're Jayden, a digital marketer working at a small publishing company called Ember Press. Part of your job is to generate promotional content for new book releases. Ember Press publishes young adult novels and is set to release a new thriller called *The Forgotten House*. You need to prepare content that will be used in emails and social media.

Here's what you'll be writing prompts for:

- Summarizing a large plot description
- Inferring customer sentiment from pre-release book reviews (so you can find a few good reviews to include in the email)
- Generating email content using the summary and positive reviews

To do these things, you'll employ techniques including:

- Using variables, delimiters, and Python's f-string feature to create more sophisticated inputs
- Instructing the model to give more specific or structured output
- Asking the model to rewrite text for tone, grammar, and spelling

Let's get to it!

SUMMARIZE THE PLOT DESCRIPTION

For this challenge you'll start with your refactored code from the end of lesson 4. Checkout the branch from that lesson and then check out a new branch to work on during this lesson.

The first thing Jayden (eh hem, you) needs to do is summarize the long-form plot description for *The Forgotten House*. [Here's the plot description](#).

Okay, we're all rooting for Mia and Alex, right?? 🥰

You need to do a few different things to prepare this text for a prompt, the first of which is to assign it to a variable.

(!) For the remainder of this lesson we'll only display the first two paragraphs of the plot description in our code examples to keep them compact, but you should use the entire passage in your code.

When assigning large blocks of text to variables in Python, it's preferable to include line breaks—like those we have in our plot description—to maintain readability within code files. These are considered “multi-line strings”. However, if you simply use single or double quotes for the variable assignment, Python will interpret the first line break as the end of

your line of code and will throw an error because it will think you forgot to add a closing single or double quote:

Even your code editor can tell something is wrong!

```
plot_description = "In the heart of the bustling metropolis of Astoria, the old Henderson House stands as a silent sentinel, its imposing facade a stark contrast to the modern skyscrapers that surround it. Once a grand mansion, it now sits abandoned, its windows broken, its once-luxurious gardens now a tangle of weeds and ivy. The house is said to be haunted, its halls echoing with the whispers of a tragic past.

When seventeen-year-old Mia Alvarez moves to Astoria with her family, she is immediately drawn to the mystery of the old house. Despite the warnings of her new friends, Mia becomes determined to uncover the truth behind the rumors that surround it."
```

Python interprets this as the end of your statement

Python can't interpret your code and throws a syntax error

```
chatbot — -bash — 90x50
Emilys-MacBook-Pro:chatbot emlaser$ python3 error.py
File "/Users/emlaser/Local Sites/chatbot/error.py", line 1
  plot_description = "In the heart of the bustling metropolis of Astoria, the old Henderson House stands as a silent sentinel,
                        ^
SyntaxError: unterminated string literal (detected at line 1)
Emilys-MacBook-Pro:chatbot emlaser$
```

Luckily, Python has a handy trick: simply surround the block of text in three double quotes, referred to as “triple quotes”, and keep the line breaks as needed:

```
plot_description = """In the heart of the bustling metropolis of Astoria, the old Henderson House stands as a silent sentinel, its imposing facade a stark contrast to the modern skyscrapers that surround it. Once a grand mansion, it now sits abandoned, its windows broken, its once-luxurious gardens now a tangle of weeds and ivy. The house is said to be haunted, its halls echoing with the whispers of a tragic past.
```

```
When seventeen-year-old Mia Alvarez moves to Astoria with her family, she is immediately drawn to the mystery of the old house. Despite the warnings of her new friends, Mia becomes determined to uncover the truth behind the rumors that surround it."""
```

Now Python understands that the line breaks within the block of text are part of the content and not a syntax error.

1. Great! Let's start by creating your `plot_description` variable which you'll add below your model variable. Between triple quotes, copy and paste the plot description linked above.
2. Now we're ready to build the prompt! We want to ask the model to summarize the plot description, which we can do by creating another variable:

```
plot_prompt = "Summarize the text below in no more than 100 words."
```

3. Cool, but... we need to include the plot description! For this we can use an **f-string**.

An **f-string** is simply a string with the letter f before the opening quotation mark, which tells Python that any expressions surrounded by curly braces within the string should be evaluated. In the case of a variable name being surrounded by curly braces, the letter f tells Python that the variable name (as well as the curly braces) should be replaced by the variable's value. Let's take a look:

```
plot_prompt = f"""
```

```
Summarize the text below in no more than 100 words.
```

```
{plot_description}  
"""
```

This statement is the same as writing:

```
plot_prompt = f"""
```

```
Summarize the text below in no more than 100 words.
```

```
In the heart of the bustling metropolis of Astoria, the old Henderson House stands  
as a silent sentinel, its imposing facade a stark contrast to the modern skyscrapers  
that surround it. Once a grand mansion, it now sits abandoned, its windows broken,  
its once-luxurious gardens now a tangle of weeds and ivy. The house is said to be  
haunted, its halls echoing with the whispers of a tragic past.
```

```
When seventeen-year-old Mia Alvarez moves to Astoria with her family, she is  
immediately drawn to the mystery of the old house. Despite the warnings of her  
new friends, Mia becomes determined to uncover the truth behind the rumors that  
surround it.
```

```
"""
```

However, the f-string makes the `prompt` variable much more readable and helps to reduce duplicate code.

4. You're almost there. Let's say you want to add additional instructions for the model in your prompt, which you add after the plot description variable. How would the model know where the description ends and your additional instructions begin? All you said at the beginning of the prompt is "the text below":

```
plot_prompt = f"""
Summarize the text below in no more than 100 words.
```

```
{plot_description}
```

```
Write this as one paragraph.
```

```
"""
```

Based on those instructions, the model would include "Write this as one paragraph" in the text to summarize. There is another trick we can use to help the model understand the exact part of the instructions it should consider the text to summarize. We can surround `{plot_description}` in special characters and tell the model to only refer to the text within those special characters. We'll use the characters `<` and `>` in both the prompt and around `{plot_description}`:

```
plot_prompt = f"""
Summarize the text below, in between < and >, in no more than 100 words.
```

```
<{plot_description}>
```

```
Write this as one paragraph.
```

```
"""
```

5. Now the model *should* understand that the text to summarize is only the text between `<` and `>`.
6. Next, in your messages array, replace `user_input` with `plot_prompt`:

```
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": plot_prompt}
]
```

7. Finally, comment out the line where you define the `user_input` variable (just above where the model variable was declared.) We want the full script to run as soon as we execute the file in the command line, and we don't need any user input for this challenge!

```
# user_input = input("\nAsk something...\n\n")
```

8. Save your file and run your script. You should have a short summarization of the plot description!
9. Now you can try adding different instructions for the model. For example, you could ask the model to make the summarization more exciting, since you want to drum up interest ahead of the book release:

```
plot_prompt = f"""
Summarize the text below, in between < and >, in no more than 100 words.
```

```
<{plot_description}>
```

```
Write this as one paragraph and make the summarization exciting. This text will be
used to promote the launch of a new book.
```

```
"""
```

10. Next, update the API call's variable name from `response_for_user` to `book_summary`.
11. And update the print statement at the bottom of the file to show the `book_summary` in the command line.
12. Save and run the script again. Did the output change based on your updated instructions??
13. Now you have copy that could be used as the basis for an Instagram or Facebook post. How handy for a digital marketer!!
14. Try a few different instructions before moving on to the next exercise, and don't be scared to get creative: maybe you want it written from the point of view of someone

from the Henderson family, or maybe you want the summarization to be longer or shorter. Can you figure out how to include the title of the book and ask the model to refer to it in its summarization??

INFER CUSTOMER SENTIMENT FROM PRE-RELEASE BOOK REVIEWS

Next you'll take a list of book reviews provided by pre-release readers and determine the sentiment of each review. In the next exercise, you'll use the positive reviews, along with your summarization, to generate email content.

After all, what's a book release without some killer pre-release book reviews??

(!) Again, we'll just show the first two reviews here, but you should include all of them in your code! Be sure there's a comma after each review. 😊

1. First, create a dictionary called `book_reviews` below the API call for the `book_summary` and add *all* of the reviews from the plot description linked above.
2. `book_reviews = {`
 "I read The Forgotten House and found it to be average. The writing was decent, and the plot was somewhat engaging, but it didn't leave a lasting impression on me.",
 "I found The Forgotten House to be predictable and lacking in originality. The plot felt formulaic, and the characters were one-dimensional. Overall, a disappointing read."
 }
3. Next, create an empty list that will eventually hold your reviews as well as the sentiment for each review: `book_reviews_with_sentiments = []`

Finally, you'll create a for loop so you can loop through the book reviews. To refresh your understanding of for loops, refer back to the [Using Loops in Your Code](#) lesson.

The for loop will include:

- A prompt to evaluate the reviews
- A message defining the system and user roles
- A call to the API
- And a method to append both the review and sentiment to the `book_reviews_with_sentiments` array.

Let's get started!

1. Write a prompt to ask the model to determine the sentiment of the current book review as a single word, positive or negative. (Be sure to add the f-string and triple quotes.)
2. Within the loop, add a `review_messages` array so you can reference your new prompt.
3. Make the API call (you can use your existing `get_api_chat_response_message` function!) and store the response in a variable. Be sure to update the parameters to include the review's messages.
4. Append a dictionary containing the current review and its sentiment to the `book_reviews_with_sentiments` array
5. Below the print statement for the `book_summary`, add a print statement for the `book_reviews_with_sentiments` array.

This may sound like a lot, but you can do it! Go step-by-step, adding `print()` lines and running your script in the command line as needed. A few suggestions:

- Does every review also have a sentiment?
- Compare the sentiment to the review — did the model do a good job of identifying the sentiment? Is it formatted correctly? Adjust your prompt as needed.
- Does adjusting the system role content help with getting a better result?
- You'll know your for loop is working when you print `book_reviews_with_sentiments` and see something similar to this:

```
[{'review': "I read The Forgotten House and found it to be average. The writing was decent, and the plot was somewhat engaging, but it didn't leave a lasting impression on me.", 'sentiment': 'Negative'}, {'review': 'I found The Forgotten House to be predictable and lacking in originality. The plot felt formulaic, and the characters were one-dimensional. Overall, a disappointing read.', 'sentiment': 'Negative'}]
```

If you need help or want to check to make sure your code is formatted properly, take a peek at [the solution code](#).

USE THE SUMMARY AND SENTIMENT ANALYSIS TO GENERATE EMAIL CONTENT

Now that you have your book summary and you've analyzed the sentiment of the pre-release book reviews, you can build the content for your book release announcement email!

The steps here are similar to what you've already done so far in this challenge, so this time you get a list of to-dos to work through independently:

1. Create a new prompt to ask the model to generate an email, using your `book_summary` and `book_reviews_with_sentiments` data
 - a. You'll need to loop through `book_reviews_with_sentiments` to get just the positive reviews!
2. Make your API call and `print()` it so you can run your script and see the results — how does it look?
3. Adjust your prompt as needed to get the email content you want — remember, we want an exciting email that makes people want to run out and buy the book!
4. Did you ask for an email subject line? You could try asking for 10 different options for email subject lines and see what you get

When you're all done, deactivate your virtual environment. Then add and commit your changes, push the commit for this branch to your GitHub repo and enter the link below!

5.5 Single-Turn vs. Multi-Turn Conversations: What's the difference?

In your next and final challenge of this lesson, you'll transform your chatbot from a single-turn conversation into a multi-turn conversation using the assistant role.

But first... what do we mean when we say single-turn vs. multi-turn?

A **single-turn** conversation is simply one where the user gives one prompt and receives one response in return. In a single-turn conversation there is no conversation history because the conversation ends after one prompt & response iteration, called a turn. This is currently how your chatbot functions. If you run your chatbot script multiple times, it has no awareness of prompts you've given it in the past.

Go ahead and try it out! Run your chatbot script and give it your name (a prompt like, "hi i'm *your name*" will do just fine). Now run your chatbot script again and ask it what your name is. Does it know?? 🤔

A **multi-turn** conversation is one where the user can have multiple prompt & response iterations—or turns—within the same chat session, creating a more realistic conversational

experience. This also means that the chatbot can keep track of the conversation history and is able to recall information from previous prompts and take the entire conversation into consideration when giving responses.

Let's compare what single-turn vs. multi-turn looks like in pseudocode.

By now you're familiar with how your single-turn chatbot script works. Here are the steps written in pseudocode:

- prompt the user for input

- define the model and messages

- make the API call

- extract the message content and display it to the user

Now let's take a look at pseudocode for what your chatbot script will look like after you complete the next challenge and transform it into multi-turn. Don't worry about understanding this in its entirety just yet!

- create an array that will store the chat history

- while the conversation is running:

 - prompt the user for input

 - if the user types "exit", stop the loop

 - add the user's input in the chat history

 - make the API call

 - extract the message content and display it to the user

 - add the message content to the chat history

 - (now the loop starts over!)

As you can see, this works similarly to a single-turn conversation, except you'll be using a while loop to keep the conversation going AND you'll be storing the conversation history in an array and sending that to the model so it has the full conversation history to reference every time it generates a response.

Pretty neat, huh?? On to the next challenge!

5.6 Refactor Your Chatbot to be Multi-Turn With the Assistant Role

Before you get started, let's take a second to celebrate how far you've come.

The chatbot you've been creating throughout this class, while simple, is REALLY awesome. Which means that YOU are awesome! Before starting this class you had no experience with OpenAI, and after just a few lessons, you know how to connect to the OpenAI API and interact with it.

These wins are worth celebrating! 🎉

In this next challenge, you'll change roles from Jayden the digital marketer to Seren, a full stack developer who works at Ember Press.

As the company's main developer, you're responsible for the functionality of the chatbot tool that Jayden has been using. Jayden loves the chatbot, but they gave you some constructive feedback: it would be REALLY great if using the chatbot felt more like a back and forth conversation, which you can now identify as a multi-turn conversation!

(Jayden would also love for the chatbot to live in a browser, a totally reasonable request. You'll get to that later!)

Your next task is to iterate on your chatbot to make it more interactive so that Jayden and other chatbot users can have multi-turn conversations, much like a more sophisticated chatbot like ChatGPT. Here's what you'll do:

- Create a while loop to manage the conversation lifecycle (i.e. keep the conversation running until the user chooses to terminate it)
- Use the assistant role to give the chatbot conversation context
- Greet your user and ask for their name to personalize the conversation 😊

Let's get to it!

REVIEW THE PSEUDOCODE

In the last step you took a look at pseudocode for the multi-turn version of your chatbot. Here's the pseudocode for your reference throughout this challenge:

```
create an array that will store the chat history

while the conversation is running:
    prompt the user for input
    if the user types "exit", stop the loop
    add the user's input in the chat history
    make the API call
    extract the message content and display it to the user
    add the message content to the chat history
    (now the loop starts over!)
```

CREATE A WHILE LOOP TO MANAGE THE CONVERSATION LIFECYCLE

Start by downloading [the starter code](#). You'll need to spin up a virtual environment and install the openai library. If you're unsure of the steps, refer back to this lesson or check out the README.md file downloaded with the starter file..

1. First, we need an array in which to store the chat history (this can go right below your model variable):

```
model = "gpt-3.5-turbo"
```

```
chat_history = []
```

2. Next, start the while loop:

```
chat_history = []
```

```
while True:
```

3. `while True` is simply stating that the while loop should run forever until it's explicitly broken out of. This is the method by which the conversation between the chatbot and the user will continue in a back-and-forth manner. If you need a refresher on while loops, refer back to [Using Loops in Your Code](#).

4. Now you need to prompt the user for input and if their input is "exit", break out of the while loop:

```
chat_history = []
```

```
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break
```

5. The `break` statement is only used when the user has decided to stop the chat by typing "exit". And you have your handy friend `lower()` to make sure you're comparing strings of the same case (remember, "EXIT" is not the same as "exit" to Python!).

And that's it for the while loop! Now you need to add the logic to handle the multi-turn conversation details.

USE THE ASSISTANT ROLE TO GIVE THE CHATBOT CONVERSATION CONTEXT

1. Now you need to log the user's input to the chat history using `append()`. As the name implies, the `append()` function tacks whatever you give it onto the end of what already exists in `chat_history`.

You need to log both the role (user) and the user's input as the content, just as you would format the messages dictionary for a chat completions API call.

Why, you ask? Because you're going to send the entire chat history as part of the API call, so your chat history array needs to be formatted for that. You'll see that in a second!

```
chat_history = []
```

```
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    chat_history.append({
        "role": "user",
        "content": user_input
    })
```

2. Now you need to make the chat completions API call, which should look familiar by now! This is where you want to send the *entire* chat history so that the model can see the full thread and have context for the conversation:

```
chat_history = []

while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    chat_history.append({
        "role": "user",
        "content": user_input
    })

    response = get_api_chat_response_message(model, chat_history)
```

3. We're getting close! Now, display the model's response back to the user, adding "Chatbot: ", before `response` so the user knows that it's the chatbot's response:

```
chat_history = []

while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    chat_history.append({
        "role": "user",
        "content": user_input
    })

    response = get_api_chat_response_message(model, chat_history)

    print("Chatbot: ", response)
```


4. Finally, log the model's response to the chat history as a message dictionary, just like what we did for the user's input. This time you'll use the assistant role as that's the role responsible for tracking the conversation history:

```
chat_history = []

while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    chat_history.append({
        "role": "user",
        "content": user_input
    })

    response = get_api_chat_response_message(model, chat_history)

    print("Chatbot: ", response)

    chat_history.append({
        "role": "assistant",
        "content": response
    })
```

5. The moment of truth!! Save your file, head over to the command line, and run your script. You should see something like this:

A screenshot of a macOS terminal window. The title bar at the top reads "chatbot-emily — Python chatbot.py — 90x50". The terminal content shows the command "Emilys-MacBook-Pro:chatbot-emily emlaser\$ python3 chatbot.py" being executed, followed by the prompt "You: " and a cursor. The terminal has a dark background with light-colored text.

```
chatbot-emily — Python chatbot.py — 90x50
Emilys-MacBook-Pro:chatbot-emily emlaser$ python3 chatbot.py
You: 
```

6. And if you type a message and hit enter, it will start to look like this:

```
chatbot-emily — Python chatbot.py — 90x50
Emilys-MacBook-Pro:chatbot-emily emlaser$ python3 chatbot.py
You: hi i'm emily
Chatbot: Hello Emily! How can I assist you today?
You: why is the sky blue?
Chatbot: The sky appears blue because of the way sunlight interacts with the Earth's atmosphere. Sunlight is made up of a spectrum of colors, with each color having a different wavelength. When sunlight reaches the Earth's atmosphere, the shorter blue and violet wavelengths are scattered by the gases and small particles in the air, while the longer wavelengths like red, yellow, and orange pass straight through. This scattering of blue and violet light is what gives the sky its blue color.
You: 
```

If yours isn't looking like this, or if you're getting errors, go back through the instructions above line-by-line. When you're ready, type "exit" to quit the chat.

GREET YOUR USER

This is a small addition that has a big impact in terms of user experience. Add code that greets the user, informs them that they can type exit to end the chat, and asks for their name. This should happen as the first prompt for the user, and you'll want to append this information to the chat history so that the chatbot can remember the user's name.

Go ahead and try that out, and refer to [the solution code](#) if you need help!

You can test this by giving the chatbot your name, having a bit of conversation, and then asking it something like, "what's my name?". If it doesn't respond with your name, check your logic and try again.

Other things you can try:

- Practice your prompting by generating the content from the last challenge using your updated chatbot
- Have a short conversation with the chatbot and then see how far back it can remember

When you're all done, deactivate your virtual environment. Then initialize the project folder as a Git repo and add and commit your work. Create a GitHub repo for the project and push to your GitHub repository. Enter the link to your GitHub repo below and mark the lesson step as complete. You're doing fantastic, Skillcrusher!!!

Lesson 6: Tokens

6.1 What are tokens?

In this lesson, you'll learn:

- What tokens are in AI and how they work
- Why tokens are important to you as a developer
- How to manage tokens during development
- And how to add token limits to your chatbot

For a large language model to process any text input it has to be taught how to process Natural Language (aka how to understand what you mean when you write, in English, "What's the weather today?").

And in order to make sense of that string of 24 characters and a question mark the AI model has to first break the data down into the smaller units.

These units are called tokens. Tokens can represent one character, a few characters, or an entire word depending on the word.

As humans who speak English, we have our own building blocks for processing language — letters of course, but also words and phrases that we can process quickly and build into sentences and paragraphs.

For example, a lot of our words and phrases contain common letter patterns. Take "ing" for example: thing, swing, going, bingo, tinge, etc. We're able to recognize those letter patterns because they are common and familiar, allowing those of us who can read to read more quickly.

In addition, our brains have other ways of recognizing words and understanding meaning that are also all about pattern recognition. Take, as an example, this paragraph of text:

I cnduo't bvlieie taht I culod aulacly uesdtannrd waht I was rdnaieg. Unisg the icndeblire pweor of the hmuam
mnid, aocdcnrig to rseecrah at Cmabrigde Uinervtisy, it dseno't mttar in waht oderr the lterets in a wrod are,
the olny irpoamtnt tihng is taht the frsit and lsat ltteer be in the rhgit pclae. The rset can be a taotl mses and
you can sitll raed it whoutit a pboerlm. Tihs is bucseae the huamn mnid deos not raed ervey ltteer by istlef, but

the wrod as a wlohe. Aaznmig, huh? Yaeh and I awlyas tghhuot slelinpg was ipmorantt! See if yuor fdreins can raed tihs too.

Here every word has been scrambled, with the exception of the first and last letter. And yet, most English speakers can skim this paragraph and fully understand what is written. That's our brain's pattern recognition at work!

Computers obviously process data much differently than human brains do and because of this, models have their own way of interpreting our human language—you guessed it, tokens!

Tokens are common sequences of characters found in a set of text, and they bridge the gap between what we understand and input as natural language and a format that the model can understand. By breaking words into tokens, the model is able to recognize patterns not only in the letters and words themselves, but also their context and relationships to one another.

If you remember from Lesson 2, transformer models work on sequential data, like a sentence, and process an entire sequence at the same time. When they're trained, transformer models learn the relationships between tokens so they can process the input data and generate an output that's relevant and makes sense.

Tokens represent pieces of words, but the exact token-to-character ratio depends on the specific language. Since tokens aren't always a perfect fit as entire words, there is a breakdown for segmenting characters or words into tokens. (Note: \approx represents approximately equals)

In English:

1 token \approx 4 chars

1 token \approx $\frac{3}{4}$ words

100 tokens \approx 75 words

1-2 sentences \approx 30 tokens

1 paragraph \approx 100 tokens

1,500 words \approx 2048 tokens

Take a look at the following sentence:

I'm going to use AI to relaunch my career.

There are 42 characters so you could guess that there are approximately 10-11 tokens, which isn't far off from the exact answer — 12.

Different languages have different token-to-character ratios because some languages need to be broken down into smaller tokens for the model to process them. English has an approximate 1:4 token-to-character ratio, while Spanish has a ratio of approximately 1:2 token-to-character ratio (we won't fall down a linguistics rabbit hole here, but this has to do with how the Spanish language is constructed).

Why does this matter to you, the developer?

Tokens are material to you, the developer, because OpenAI charges for API usage by the token—meaning the more tokens your app uses, the bigger your bill! Another major factor affecting your cost is the version of the large language model you decide to use. For example:

gpt-3.5-turbo-0125 (the model we will use to for our work) costs the following:

Input: \$0.50 / 1M tokens

Output: \$1.50 / 1M tokens

Now you know what tokens are, and why you need to be thinking about them, we can explore how you can take into consideration token cost as you build your application!

6.2 Evaluate Your Chatbot's Token Usage

As you learned in the last step, managing token usage is a crucial part of any application that utilizes the OpenAI API. In this challenge, you'll learn two methods for evaluating and understanding token usage. You will:

- Get familiar with Tokenizer
- Evaluate the token usage of your chatbot using Tokenizer
- Evaluate your chatbot's token usage via the API response data

Now, before we begin, it's important to note that the token usage of your chatbot is very small due to the fact that it's a small app, making small requests, with only a small number of users. But, as we've discussed, this information is crucial for understanding and planning

how much your app *could* cost in the future if you were to do anything like expand functionality, expose it to more users, or integrate it into a bigger application—most of which you'll be doing in future classes!

GET FAMILIAR WITH TOKENIZER

To help developers understand how text is tokenized by their language models, OpenAI has a handy tool called [Tokenizer](#).

Tokenizer is a simple web interface where you can enter text and see the tokenized version of that text. You can also see the total number of tokens as well as a visualization of how the model tokenized the text:

The screenshot shows the OpenAI Tokenizer web interface. At the top, there are two tabs: "GPT-3.5 & GPT-4" (selected) and "GPT-3 (Legacy)". Below the tabs is a text input area containing the following text:

Many words map to one token, but some don't: indivisible.

Unicode characters like emojis may be split into many tokens containing the underlying bytes: 🙌

Sequences of characters commonly found next to each other may be grouped together: 1234567890

Below the input area are two buttons: "Clear" and "Show example". Below the buttons, there is a table showing the tokenization results:

Tokens	Characters
57	252

Below the table, the input text is displayed with each token highlighted in a different background color. The tokens are: "Many", "words", "map", "to", "one", "token,", "but", "some", "don't:", "indivisible.", "Unicode", "characters", "like", "emojis", "may", "be", "split", "into", "many", "tokens", "containing", "the", "underlying", "bytes:", "🙌🙌🙌🙌🙌", "Sequences", "of", "characters", "commonly", "found", "next", "to", "each", "other", "may", "be", "grouped", "together:", "1234567890".

At the bottom, there are two tabs: "Text" (selected) and "Token IDs".

Tokenizer will show you the number of tokens needed with each token represented as a different background color. The space between words is always the first character of a token.

The beauty of Tokenizer is that you can evaluate content for token usage without having to actually make API calls and use any *real* tokens. This means that as a developer, you can

test out the functionality of your app and understand how it may scale without spending any money on tokens.

However, to make things easier, in this challenge you're going to interact with your script and paste parts of the conversation into Tokenizer so you can see how token usage builds the longer the conversation gets.

Before you start, let's review how OpenAI charges for token usage. As you know by now, the most basic way in which a language model works is that it takes an input, processes that input, and then delivers an output. OpenAI charges for inputs and outputs at different rates. Let's use the gpt-3.5-turbo-0125 model as an example (this is the version you're using in your script):

gpt-3.5-turbo-0125	
Input	Output
\$0.50 / 1M tokens	\$1.50 / 1M tokens

As of March 2024.

(!) If you look through the OpenAI documentation, you'll see a few different terms used for input and output:

Input: You may also see request, prompt, or context

Output: You may also see response, generated, or completion

To add to the confusion, you may see references to a "context window", which is the token limit when you combine both the input token usage and output token usage.

EVALUATE THE TOKEN USAGE OF YOUR CHATBOT USING TOKENIZER

Start up your environment so that you can run the script from [the last challenge](#). Run your script and interact with it a few times so that you have 4-5 conversation turns.

Next, copy the conversation and paste it into Tokenizer to see how many tokens you've used so far. Don't include the initial "Hello there..." message, as that's just the greeting to the user and is not included in the chat history.

Copy the entire conversation and paste it into Tokenizer.

```
chatbot-emily — Python chatbot.py — 100x60
Emilys-MacBook-Pro:chatbot-emily emlaser$ python3 chatbot.py
Hello there, I'm your helpful chatbot! What's your name? Emily
Chatbot: Hello Emily! How can I assist you today?
You: can you tell me how many books stephen king has written?
Chatbot: Stephen King has written over 60 novels and more than 200 short stories. He is an incredibly prolific and successful author in the horror and supernatural fiction genres.
You: what are his five most popular novels?
Chatbot: Some of Stephen King's most popular novels include:

1. "It" - A horror novel about a group of children who are terrorized by an evil entity that takes the form of a clown named Pennywise.
2. "The Shining" - A psychological horror novel about a man who becomes the winter caretaker of an isolated hotel and slowly descends into madness.
3. "Carrie" - King's first published novel, about a high school girl with telekinetic powers who seeks revenge on her classmates.
4. "Misery" - A psychological thriller about an author who is held captive by an obsessed fan.
5. "The Stand" - A post-apocalyptic novel about a group of survivors who must rebuild society after a deadly pandemic wipes out most of the population.

These are just a few of Stephen King's most popular and well-known novels.
```

How many tokens is your conversation? My conversation, seen in the screenshot above, is 256 tokens.

Next, continue the conversation so you have an additional 4-5 conversation turns (or even more!). Copy the conversation again and paste it into Tokenizer. Remember—in your *script*, you're sending the whole conversation in every API call so that the model has the full conversation history and context, so we want to see how the token usage increases as the conversation grows.

Copy the entire
updated
conversation
and paste it
into Tokenizer.

```
chatbot-emily — Python chatbot.py — 100x60
Emilys-MacBook-Pro:chatbot-emily emlaser$ python3 chatbot.py
Hello there, I'm your helpful chatbot! What's your name? Emily
Chatbot: Hello Emily! How can I assist you today?
You: can you tell me how many books stephen king has written?
Chatbot: Stephen King has written over 60 novels and more than 200 short stories. He is an incredibly prolific and successful author in the horror and supernatural fiction genres.
You: what are his five most popular novels?
Chatbot: Some of Stephen King's most popular novels include:

1. "It" - A horror novel about a group of children who are terrorized by an evil entity that takes the form of a clown named Pennywise.
2. "The Shining" - A psychological horror novel about a man who becomes the winter caretaker of an isolated hotel and slowly descends into madness.
3. "Carrie" - King's first published novel, about a high school girl with telekinetic powers who seeks revenge on her classmates.
4. "Misery" - A psychological thriller about an author who is held captive by an obsessed fan.
5. "The Stand" - A post-apocalyptic novel about a group of survivors who must rebuild society after a deadly pandemic wipes out most of the population.

These are just a few of Stephen King's most popular and well-known novels.
You: can you tell me more about the novel "the stand"
Chatbot: "The Stand" is a post-apocalyptic novel written by Stephen King. It was first published in 1978 and has since become a classic in the genre. The story is set in a world devastated by a deadly pandemic known as "Captain Trips," which wipes out the majority of the population.

The novel follows a group of survivors who are immune to the virus and must navigate the dangers of a world without order or civilization. As they struggle to rebuild society, two factions emerge: one led by the benevolent Mother Abigail and the other by the nefarious Randall Flagg, a dark and powerful figure who represents chaos and destruction.

"The Stand" explores themes of good versus evil, the resilience of the human spirit, and the consequences of societal collapse. It is a compelling and thought-provoking story that has captivated readers for decades.
You: is the novel set in derry maine like a lot of his other novels?
Chatbot: No, "The Stand" is not set in Derry, Maine like many of Stephen King's other novels. The story takes place in various locations across the United States, including Texas, Nebraska, Colorado, and Las Vegas. However, Derry, Maine is a fictional town that frequently appears in King's works, such as "It" and "Insomnia."
You: tell me about the tv series adaptation of the book
Chatbot: "The Stand" has been adapted into a television miniseries twice. The first adaptation aired in 1994 and starred Gary Sinise, Molly Ringwald, and Rob Lowe, among others. This miniseries follows the basic plot of the novel, but due to time constraints, some characters and storylines were condensed or omitted.

The second adaptation is a miniseries that premiered in 2020 on CBS All Access. This version is a more faithful adaptation of the novel and features a star-studded cast including James Marsden, Whoopi Goldberg, and Alexander Skarsgård. The series expands on the characters and storylines from the novel and delves deeper into the themes of good versus evil and the resilience of humanity.

Both adaptations have received mixed reviews from critics and audiences, with some praising the performances and faithfulness to the source material, while others have criticized certain changes or pacing issues. Overall, "The Stand" remains a popular and compelling story that continues to captivate audiences in its various adaptations.
```

With just a few more conversation turns, my token usage grew to 768.

Let's do a little math using the token cost above.

First let's determine the cost of one token:

$\$1.50 / 1,000,000 = \0.0000015

Then we can use that to determine the cost of my conversation:

$\$0.0000015 \times 768 = \0.0011520

(You might be thinking, “Hey, you’ve said I don’t need to be good at math to be a developer!” That holds true, and we promise, this is the most math you’ll be doing in this class!!)

Even though the total amount may seem insignificant, understanding this cost is a critical skill for a developer to have. One big reason for this is that your chatbot’s usage may grow over time, and you need to understand how the associated token usage cost will scale. How much token usage would there be if 1,000 people used your chatbot every day for a month? 10,000? 1,000,000? What if those 1,000,000 used your chatbot every day for a year??

Complete this process—continuing your conversation and then evaluating your input token usage—a few more times to get a better understanding of how quickly token usage scales as the input volume increases.

EVALUATE YOUR CHATBOT’S TOKEN USAGE VIA THE API RESPONSE DATA

You'll be working with the code from the challenge in which you updated your chatbot to be multi-turn. In the command line, navigate to the chatbot directory. Then checkout the main branch.

Open the chatbot directory in your text editor and, in the chatbot.py file, locate the code that looks like this:

```
response = client.chat.completions.create(  
    model = model,  
    messages = messages  
)
```

Below that line, enter another line to print `response`. If you run your script and enter an input, you should see something like this:

```
ChatCompletion(id='chatcmpl-979GvAl3psXTfSVwGbWpQ4JbpKaLT',  
choices=[Choice(finish_reason='stop', index=0, logprobs=None,  
message=ChatCompletionMessage(content='Hi Emily! How can I assist you today?',  
role='assistant', function_call=None, tool_calls=None))], created=1711490965,  
model='gpt-3.5-turbo-0125', object='chat.completion', system_fingerprint='fp_3bc1b5746c',  
usage=CompletionUsage(completion_tokens=10, prompt_tokens=8, total_tokens=18))
```

The part we're interested in is toward the end, `usage`:

```
usage=CompletionUsage(completion_tokens=10, prompt_tokens=8, total_tokens=18)
```

This object contains everything we want to know about the token usage (note that the values you see may be slightly different):

```
completion_tokens=10,  
prompt_tokens=8,  
total_tokens=18
```

You can see here that the input (aka `prompt_tokens`) was 8 tokens, the output (aka `completion_tokens`) was 10 tokens, and the total number of tokens used was 18.

Now you can use these usage numbers to get a more accurate understanding of cost, referring back to the usage cost chart:

gpt-3.5-turbo-0125	
Input	Output
\$0.50 / 1M tokens	\$1.50 / 1M tokens

1 input token = \$0.0000005

1 output token = \$0.0000015

8 input tokens x \$0.0000005 = \$0.0000040

10 output tokens x \$0.0000015 = \$0.0000150


Total token usage = \$0.0000190

Go ahead and run your script a few times so you can calculate a more accurate cost as the conversation grows.

When you're done, be sure to delete the `print` line you added for this exercise.

Add a screenshot of your terminal window below and mark this lesson step complete!

6.3 Tactics for Managing Token Usage

Now you have a basic understanding of what tokens are, how they work, and why they exist in generative AI AND you've gotten a feel for your chatbot's token usage (even if the number of tokens used is relatively small for now). 

But what does this all mean for your chatbot script and other projects you'll build using OpenAI? Being able to estimate the token usage of your web applications only goes so far. As a developer, you also need levers to actually *control* the token usage of your app while still maintaining helpful and engaging conversations.

And that's what the rest of this lesson is all about! Let's talk more about how tokens are tracked and managed in code as well as what you can do as a developer to manage the number of tokens used.

As a reminder, understanding token usage is important for a few different reasons:

- The cost of token usage can grow quickly depending on how sophisticated your chatbot is, how many users your chatbot has, and how long those users chat.
- The speed at which API calls process depends on how many tokens are being used — the more tokens you use, the slower your API calls will process.
- Your API call can fail if you exceed the model's token limit. The model you're using in this class, gpt-3.5 turbo, has a limit of 16,385 tokens for input and output combined. That may sound like a lot, but if your user chats for a long time, it's likely that your chatbot will hit this limit.

With that in mind, here are some tactics and best practices for managing token usage.

Give the model (and user) example prompts for context and expected behavior.

You saw this tactic back in the [Understanding Message Roles](#) step in lesson 5. Giving the model example prompts and responses via the user and assistant roles helps the model to understand expected behavior:

```
completion = client.chat.completions.create(  
    model = "gpt-3.5-turbo",
```

```

messages=[
    {"role": "system", "content": "You're a friendly and helpful assistant."},
    {"role": "user", "content": "Tell me a joke."},
    {"role": "assistant", "content": "Sure! Here's an example of a joke: Why don't scientists trust atoms? Because they make up everything!"},
    {"role": "user", "content": "That's funny! Can you tell me another one?"},
    {"role": "assistant", "content": "Of course! Here's another joke: Why did the math book look sad? Because it had too many problems."}
]
)

```

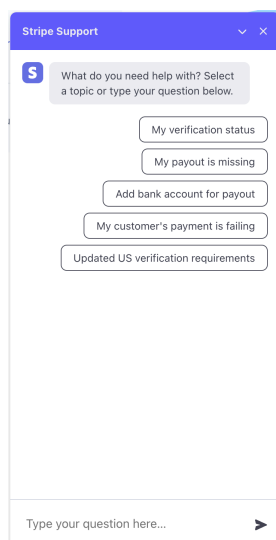
You can also give the user instructions for how they should write their prompts:

```
input("Hello there, I'm your helpful chatbot! Type exit to end our chat. Please keep your prompts short and concise. ")
```

Both of these tactics help to mitigate the likelihood of the user needing to send follow-up prompts if the model didn't understand their initial prompt.

Offer prompt options to users as their first interaction.

If you've used any kind of online customer service chatbot, you've likely seen this tactic:



This is the chatbot available to Stripe business customers.

Instead of the model having to parse a freeform prompt, it can instead process a clear and concise pre-written prompt. In the Stripe Support example above, it's likely that the chatbot already has answers to these questions ready to serve to the user thanks to caching. Which leads us to our next tactic!

Cache the answers to frequently asked questions and prompts.

This is quite common in online customer service chatbots. Over time, as the number of users who engage with the chatbot increases, patterns will emerge in the types of prompts being used. The answers to the most frequently asked questions can be cached and easily served to users without the chatbot even having to send an API request to the model, which helps to reduce costs!

Limit the number of characters the user can type for their prompt.

This one is quite simple—set a maximum number of characters that the user can type into the prompt input field. If you use this tactic, you should include a small line of text informing the user that they only have, for example, 100 characters to write their message. With this tactic, users won't be able to type incredibly long prompts which unnecessarily use up tokens.

What is your question?

The user won't be able to type anything once they hit they 100-character mark.

Please limit your question to 100 characters or less.

Helpful message for the user.

This may seem like it goes against the prompt best practices outlined in [this step](#), but limiting the number of characters the user can type will actually encourage them to think more about how to be clear and concise.

Filter the prompt to remove unnecessary words.

Remember back in the [Give Your Chatbot Some Personality challenge in lesson 5](#), where you created an array of keywords to identify a user's prompt as a question? You can use a similar tactic to remove words that don't add clarity to the user's prompt to reduce the number of characters (and thus the number of tokens). Let's look at a simple code example:

```

# Define a list of unnecessary words
unnecessary_words = ["please", "can you", "could you", "would you", "kindly"]

# Split the prompt into individual words
words = prompt.lower().split()

# Remove unnecessary words
stripped_prompt = ' '.join([word for word in words if word not in unnecessary_words])

# Remove punctuation and extra spaces between words
stripped_prompt = re.sub(r'\W+', '', stripped_prompt).strip()

return stripped_prompt

```

In the example above, the script strips out any unnecessary words from the prompt, and then strips out any spaces or punctuation, all before the prompt is passed to the model. This helps to reduce the number of characters in the prompt while still maintaining the user's intent.

Limit the output / response length.

Another simple tactic that can be easily implemented is using the `max_tokens` parameter in the Chat Completions API request. As you learned in the [Chat Completions API Endpoint Cheatsheet](#), `max_tokens` can be used to limit the number of tokens the model uses when generating a response:

```

response = client.chat.completions.create(
    model = "gpt-3.5-turbo",
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello"}
    ],
    max_tokens = 250
)

```

In this code example, the model will not exceed 250 tokens when generating a response.

Monitor token usage.

Monitoring the token usage of your chatbot—and any generative AI application you build in the future—is one of the best things you can do to understand the cost of your application and how it may scale over time. Many of the tactics listed above are great for reducing token usage. But you need hard usage data to really understand how users interact with your app and the associated long-term cost implications.

The simplest way to monitor usage is to log the number of tokens used during each chatbot session by appending data to a log file when the user exits the chat.

Ready to avoid going broke today?? In the next two challenges you'll employ a few of these tactics to manage the token usage of your chatbot!

6.4 Limit Your Chatbot's Token Usage

Now that you've learned some tactics for managing token usage, it's time to apply this new knowledge to your chatbot!

Again, remember that your chatbot's token usage is currently very small. This challenge is about reinforcing the skills you need to understand and plan how much your app *could* cost in the future.

Here's what you'll do in this challenge:

- Install the tiktoken module
- Add token limits to your chatbot

Here we go!

INSTALL THE TIKTOKEN MODULE

Before you start, make a new branch of the chatbot-v2 directory from Lesson 5, Step 6 for this challenge.

In this challenge you'll be using tiktoken, a tokenizer tool optimized for use with OpenAI. The premise of this tool is the same as Tokenizer, but it has two major benefits: you can utilize the tool directly in your codebase, and you can evaluate token usage *before* making API calls.

First, install the tiktoken Python module. In the command line on both Mac and Windows run:

```
pip install tiktoken
```

Once that finishes running, you can run the `pip list` command to confirm that tiktoken successfully installed.

Next, hop over to your chatbot.py script. At the very top of the file, beneath the import of openai, import tiktoken:

```
import tiktoken
```

In order for tiktoken to properly encode prompts—which simply means converting the prompts to a format understood by computers—it needs to know what model you’re using in your script. This can be determined by using one of tiktoken’s built-in functions, `encoding_for_model()`. This function takes the model as a parameter and sets the proper encoding.

Directly below the line of code in which you set your `model` variable, add another line of code:

```
encoding = tiktoken.encoding_for_model(model)
```

Next, add a print statement to print out the `encoding` variable. When you run your script in the command line, you should see this:

```
<Encoding 'cl100k_base'>
```

This is the correct encoding for your model, so you’re ready to start using tiktoken in your chatbot script. Let the fun begin!

The easiest way to count the number of tokens in a prompt is to use tiktoken’s `encode()` function. This function takes the prompt as an argument and returns a list of token integers. You can think of these integers as unique IDs for tokens in the English vocabulary.

Directly above your `chat_history.append()` block of code, add a line of code to encode the user's input. This line should be indented the same level as the `if` statement above it:

```
user_input_encoded = encoding.encode(user_input)
```

Now add a line of code below to print `user_input_encoded`. When you run your script, to be sure you're submitting multiple tokens, respond to "What's your name?" with a full sentence like "My name is *your name*". The response will be a list of token integers that represent your prompt, "My name is *your name*":

```
[44, 3092, 836, 374, 17329, 1570]
```

Now you need to actually *count* the number of tokens, because these integers don't do you much good on their own.

But you're not going to actually *count* the number of tokens, right? You're a developer so you'll use the built-in Python function `len()`, which returns the number of items in a list.

Replace `user_input_encoded = encoding.encode(user_input)` with:

```
token_count = len(encoding.encode(user_input))
```

You can see that you're still utilizing the `encode()` function, but you're now wrapping it in the built-in Python function `len()`.

Now you have a very simple way of checking token usage *before* making API calls!

Next, update the print statement to print `token_count` and run your script again. Instead of a list of integers, you'll see the total number of tokens.

(!) It's important to know that this very straightforward method of counting tokens using tiktoken does not work for every OpenAI model. Some models require more logic to get an accurate count of tokens. We won't go into detail with that logic here because it's outside the scope of this class!

ADD TOKEN LIMITS TO YOUR CHATBOT

By now you have not one, not two, but THREE different ways of evaluating the token usage of your chatbot. How versatile your skillset is becoming! 🍌

You still have one pesky little task, which is to check the number of tokens within a prompt before sending it to the model and then respond accordingly.

The main reason to do this is to make sure the user's prompt does not exceed the maximum token limit of the Chat Completions API endpoint. Using Tiktoken to programmatically count tokens is the best method to do this because you can determine whether the prompt exceeds the model's capacity *before* you send the API request. And you can assess this without having to use any external tools resulting in cleaner, better-functioning code.

According to [OpenAI's documentation](#), the maximum number of tokens in a context window—which is the number of tokens when you combine both the input token usage and output token usage—is 16,385. We also know from the documentation that the maximum number of output tokens is 4,096. Therefore, with a bit of math (sorry, math again!) we know that the maximum number of input tokens is 12,289.

So, a simple check you can do is to count the number of tokens in the user's prompt and make sure that it does not exceed 12,289. And if it does, you need to inform the user to shorten their prompt!

Right below where the encoding variable was declared, create a new variable set to the model's input token limit:

```
token_input_limit = 12289
```

Now, right below your `token_count` variable inside your while loop, add a new block of code that checks the user's prompt against the input token limit and informs the user if their prompt is too long:

```
if (token_count > token_input_limit):  
    print("Your prompt is too long. Please try again.")  
    continue
```

The `continue` keyword tells the while loop to start a new iteration of the loop, which will tell the user to enter another prompt.

You may be wondering two things: Why the heck would anyone enter that much text into the chatbot? And how the heck am I going to test this thing?? 🤔

There are answers to both!

For the first question: Users can be very unpredictable, so as a developer it's good practice to try to account for all the nutty things they may try to do with your chatbot. Imagine you're an editor at Ember Press, and you want to use the chatbot to generate a plot description for another new book. You open the PDF manuscript and copy the ENTIRE contents of the PDF into the chatbot and hit enter. It's safe to say that this would exceed the model's input token limit, and your chatbot would likely crash. Your script should account for this unexpected behavior.

For the second question: You can temporarily adjust the value of your `token_input_limit` variable in order to test your script's logic. Try it out by changing `12289` to `2`, saving the file and running your script again. Interact with your script a few times with a single word and then with a longer sentence. Considering your input token limit is set to `2`, does the chatbot behave as expected?

One thing you may consider doing is setting `token_input_limit` to a value that is still large but lower than `12289` to limit token usage even more.

When you're done, comment out the statement that prints the total number of tokens. Deactivate your virtual environment. Add and commit your changes and push this branch to your GitHub repository. Then enter the link below.

If you have any issues running your code, compare it with [the solution code](#). Small differences in indentation is often the issue when writing Python code. 😊

6.5 Log Your Chatbot's Token Usage

In the final challenge of this lesson, you'll log your chatbot's token usage to a log file. A log file keeps track of information about the running of your program. In this case, the log file will show the HTTP response and the total number of tokens used each session.

As you learned in the [Best Practices for Managing Token Usage](#) step, logging your chatbot's token usage is the best way to track usage data over time to understand how the cost may scale. Having a running log of usage means you can export this data to generate usage reports or even create charts using Matplotlib. The finance team at Ember Press will LOOOOOVE you 😊

This is another challenge where we aren't going to guide you step-by-step because we know you can do it! And may we add a gentle reminder that you should not expect to get this right the first time. This is meant to challenge you to integrate the things you've been learning throughout class so far and expand your Python knowledge.

Here are some hints to get you started:

- Create a branch off the last challenge. Creating a branch will allow you to experiment while preserving your code.
- For this task you should use [Python's logging module](#). Logging comes standard in Python's standard library of tools, so you don't need to install it. You only need to import it at the top of your Python script.
- You want to log the *total* token usage of each chat session, so you'll need to append the usage to a variable in each while loop iteration
- Refer to the [API Response Breakdown](#) section of your Chat Completions API Endpoint cheatsheet for a refresher on how you can determine the total number of tokens in each request
- You'll only write to the log once, when the user decides to end the chat session by typing "exit"
- Make sure you include the date when you log the chat session's token usage (you'll need this for reporting purposes)
- Bonus task: Can you log the input and output token usage in addition to the total token usage??

You got this, Skillcrusher! Take it one task at a time and refer to the [logging documentation](#) for help. When you're ready, compare your code to [the solution](#). Your code doesn't need to match - you may have approached solving the problem a different way (which is perfectly fine!) It will expand your understanding of how to code.

When you're all done, add and commit your work, push this branch to your GitHub repository, and enter the link below.

Lesson 7: The Assistants API

7.1 What is the Assistants API?

In this lesson you will learn:

- What the Assistants API is and how it differs from the Chat Completions API
- Why you should use the Assistants API
- How the Assistants API works
- And you'll start building your first AI assistant

The Assistants API is a new OpenAI API endpoint that can be used to interact with OpenAI's models. It allows you to build AI assistants within your web applications.

What IS an assistant? An assistant is an AI tool built for a specific purpose or use case, for example a personal financial assistant that can access your banking data and help you create a budget, or a book recommendation app that suggests new books based on your personal reading history.

You can think of an assistant as a really powerful chatbot that can do things like use multiple models, maintain long conversations, reference data that's not accessible via the open internet, respond to complex user queries, and run functions.

"BUT WAIT. Isn't this what I already built with the chat completions API??"

Yes! And no.

The chatbot you've been building is really good at accepting a prompt, sending that prompt to one of ChatGPT's LLM models, and extracting the response. It's also good at understanding context because your script logic includes the entire conversation history with each API request.

However, if you want your chatbot to be able to handle more complex tasks like accessing restricted data to enhance responses—something you'll be doing throughout this course—the assistants API is a better choice.

Let's review some of the major differences between the two endpoints.

API Requests & Responses

Chat Completions API: Sending an API request is quite simple with the use of `.chat.completions.create()`, and the response object is straightforward. Extracting the message from the response is easy with the help of `.choices[0].message.content`.

Assistants API: API request and response management is more complex with the Assistants API, requiring the use of threads, messages, and runs. You'll learn much more about these throughout this lesson!

Conversation Management

Chat Completions API: With this endpoint, each API request must contain the entire chat history in order for the model to understand context. This makes this endpoint better for shorter conversations.

Assistants API: The assistants API endpoint is able to maintain extended, contextual conversations with the help of threads, and truncates the messages as needed—no failed API requests due to token limits!

Capabilities

Note: Capabilities are simply features that allow you to customize how the endpoint handles your request or returns data.

Chat Completions API: This endpoint has one capability—known as a tool—you can utilize, namely function-calling. Its capabilities end there.

Assistants API: The assistants API endpoint has three capabilities—again, known as tools—available for use: function calling, code interpreter, and knowledge retrieval. We'll review these in more detail in a later step!

Use Cases

Chat Completions API: The chat completions API endpoint is ideal for short conversations that don't require a lot of context in order for the model to give an accurate answer. For example, a chatbot built with the chat completions API could accept the prompt "What can I make for dinner with kale, tomatoes, chicken sausage, and pine nuts?" and give several recipe recommendations.

Assistants API: The assistants API endpoint on the other hand is ideal for longer conversations requiring detailed context management and for conversations that require custom functionality. Continuing our “What can I make for dinner...” example above, an assistant that knows details about your overall health and nutrition needs could take those things into account and suggest meals that are catered to your personal and unique requirements. It could even be aware of meals you’ve tried in the past and whether or not you liked them.

Alright, before I send you off to start playing with the Assistants API, I need to issue a disclaimer!

The assistant API is very new, it was released in November of 2023. And as a result it doesn’t have a ton of documentation or tried and true use cases, YET.

Developers are still learning how to best leverage its capabilities and they’re still figuring out when and why and how to use the assistants API instead of the chat completions API.

Our goal here is to get you the student in on the ground floor of this emerging technology so that you have a better understanding of what all OpenAI can do, and we believe that having this knowledge will better prepare you for the future growth of the OpenAI API.

And with that, it’s time to move on to the next step and dig into how the assistants API works.

7.2 How Assistants Work: Anatomy of an Assistant

Now you know that an assistant is an AI tool built for a specific purpose or use case. Creating an assistant using the Assistants API is more complex, requiring the use of:

- The assistant itself
- Threads
- Messages
- Runs (and run steps)

Each of these is an object that represents a different aspect of an assistant’s functionality. Let’s define in more detail what each of these objects represents.

ASSISTANTS

An assistant is an AI tool built for a specific purpose or use case. An assistant utilizes threads, messages, and runs via the Assistants API.

An assistant has a name, model specification and instructions for how it should behave. The assistant can manage any number of threads independently.

THREADS

A thread is a conversation between the user and the assistant. A thread holds all the messages contained in the conversation history, however, it will automatically truncate the oldest messages in order to fit the thread into the model's context, or token limit. A thread is specific to one user and is a single conversation session.

MESSAGES

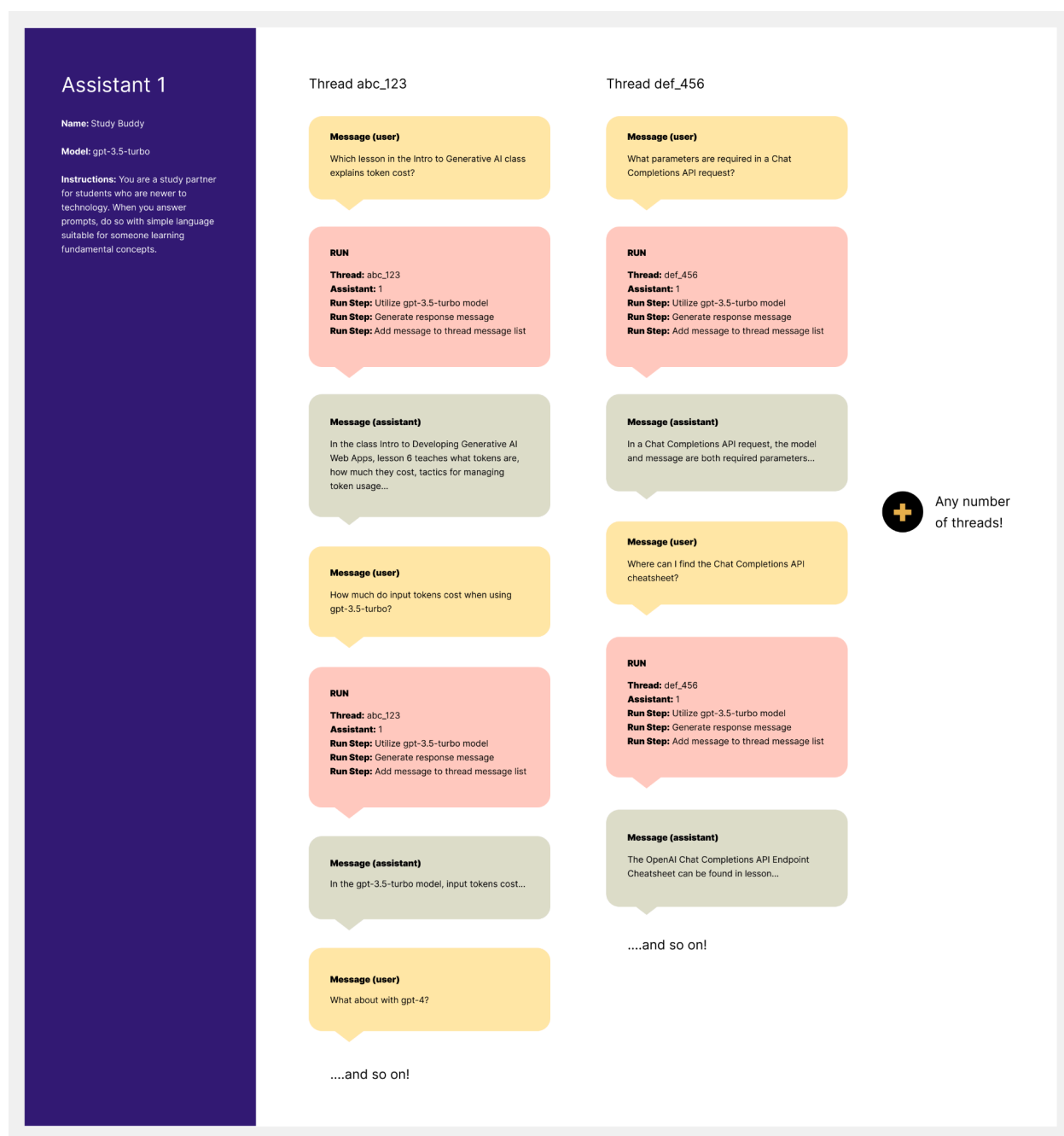
A message is exactly what it sounds like—one message created by either the user or the assistant. A message is stored in its associated thread.

RUNS (AND RUN STEPS)

A run is where the magic happens! After receiving a user message, the assistant takes its own configuration and the entire thread and then executes tasks using the model and any other specified tools. This is an on-steroids version of sending a prompt to a model as you do using the chat completions API point.

A run step is one action the assistant took within the run, for example utilizing the model, utilizing a tool, or creating the response message. Each run typically contains multiple steps.

Now, let's look at a graphic visualizing how an assistant works using these objects:



As you can see in this graphic, an assistant can manage multiple threads independently. Each thread starts when the user submits a message. The assistant takes that message as well as its own configuration and creates a run. Each run contains multiple steps to process the request using the model and any specified tools. When the run is complete, the response message is displayed to the user. And the cycle repeats!

In the next step you'll do a deep dive into the code behind all of this functionality.

7.3 How Assistants Work: Breakdown of the 4 Objects

In the last step, you were introduced to the four objects that come into play when creating assistants:

- The assistant itself
- Threads
- Messages
- Runs

Let's dig into how to work with each of these objects in code, including how to make some common API requests and how to extract data from the responses.

(!) This API is in beta! As you work through the rest of this lesson, you'll see that all the API calls include the word beta. This is because the Assistants API endpoint is new and not yet considered a fully-functional part of the OpenAI API. There is nothing bad or wrong about this, it just means that you're on the cutting edge of this new technology ✨

Furthermore, because this API is under heavy development by OpenAI, we've only included the most crucial information in this lesson. Refer to OpenAI's [Assistants API reference](#) for more information and to keep up with the latest developments!

ASSISTANTS

When developing an assistant, it must first be created. When creating an assistant, it's best practice to name it, select a model, and give it basic instructions that describe how the assistant should work (very similar to the system user role in the Chat Completions API endpoint):

```
assistant = client.beta.assistants.create(  
    name = "Study Buddy",  
    model = "gpt-3.5-turbo",  
    instructions = "You are a study partner for students who are newer to technology. When  
you answer prompts, do so with simple language suitable for someone learning  
fundamental concepts.",  
    tools=[]  
)
```

Here you can see that the assistant is named “Study Buddy,” is configured to use the gpt-3.5-turbo model, and has been given a basic description of what its purpose is and how it should answer prompts. The `model` parameter is the only required parameter.

You’ll also notice a `tools` parameter, which is a list of tools that the model should utilize when generating responses. There are three possible values for this parameter: function calling, code interpreter, and knowledge retrieval. Let’s briefly define these:

- **Function-calling:** This tool allows you to integrate custom functions into the assistant, which is useful for expanding and optimizing the overall functionality of your application. For example, you could write a function that extracts lesson topics out of a prompt like “which lesson in the Intro to Generative AI class explains token cost?” so that the assistant better understands what kind of response the user wants.
- **Code Interpreter:** This tool can write and execute Python code in a sandbox environment, which is useful for solving complex math and coding problems. It can also analyze large volumes of data from files.
- **Knowledge retrieval:** This tool extends the assistant by supplying it with data from outside the model, like personal data or proprietary business information.

`assistants.create()` returns an object that looks like this:

```
Assistant(  
  id="asst_mKqxyceU1VvAhCm9ljAewEN6",  
  created_at=1712675992,  
  description=None,  
  file_ids=[],  
  instructions="You are a study partner for students who are newer to technology. When  
you answer prompts, do so with simple language suitable for someone learning  
fundamental concepts.",  
  metadata={},  
  model="gpt-3.5-turbo",  
  name="Study Buddy",  
  object="assistant",  
  tools=[]  
)
```

To refer to the assistant's unique identifier property, you would use `assistant.id`.

THREADS

When a thread is first created, it's simply an empty container that will eventually hold the conversation between the user and the assistant. Creating a thread is simple:

```
thread = client.beta.threads.create()
```

That's it! `threads.create()` returns an object that looks like this:

```
Thread(  
  id="thread_yMHG8Ca2UpvBITxUO2uvl3NC",  
  created_at=1712685904,  
  metadata={},  
  object="thread",  
)
```

The most important property here is `thread.id`. This is the thread's unique identifier that we'll reference in subsequent steps.

MESSAGES

The most common things you'll do with messages are create messages in an existing thread and retrieve a list of messages from a thread. This is how you create a message:

```
message = client.beta.threads.messages.create(  
  thread_id = thread.id,  
  role = "user",  
  content = "Which lesson in the Intro to Generative AI class explains token cost?"  
)
```

`threads.messages.create()` has three required parameters: `thread_id`, `role`, and `content`. `thread_id` defines what thread the message belongs to. `role` and `content` probably look pretty familiar! These are used to define the role and the message itself, similar to how a chat completions API request is configured. Normally `content` would be a prompt from the user, but here we've hardcoded a message for demonstration purposes.

`threads.messages.create()` returns an object that looks like this:

```
ThreadMessage(  
  id="msg_a5vpEivZskEzanWyR0rPvQ41",  
  assistant_id=None,  
  content=[  
    MessageContentText(  
      text=Text(  
        annotations=[],  
        value="Which lesson in the Intro to Generative AI class explains token cost?",  
      ),  
      type="text",  
    )  
  ],  
  created_at=1712693049,  
  file_ids=[],  
  metadata={},  
  object="thread.message",  
  role="user",  
  run_id=None,  
  thread_id="thread_yMHG8Ca2UpvBITxUO2uvI3NC",  
)
```

This object contains lots of information about the message object. The most important property is `thread_id`—it matches the ID of the thread that the message belongs to.

This is how to retrieve a list of messages from a thread:

```
thread_messages = client.beta.threads.messages.list(  
  thread_id = thread.id  
)
```

This call has only one required parameter, `thread_id`. This is the ID of the thread for which you want to retrieve a list of messages.

`threads.messages.list()` returns an object that looks like this:

```

SyncCursorPage[ThreadMessage](
  data=[
    ThreadMessage(
      id="msg_DI45Bp0aNdaByyoWMIpj3pot",
      assistant_id="asst_GVibqVcflbIQAQMtOaBas6U5",
      content=[
        MessageContentText(
          text=Text(
            annotations=[],
            value='The lesson that explains token cost in the Intro to Developing
Generative AI Web Apps is called "Tokens." In this lesson, you will learn about the concept
of tokens and how they are used in generative AI models.',
          ),
          type="text",
        )
      ],
      created_at=1712772655,
      file_ids=[],
      metadata={},
      object="thread.message",
      role="assistant",
      run_id="run_U3BUrHeu1fYQEbLon4cMm4hS",
      thread_id="thread_pSXweFuKtYQA3hcpJs9ashQp",
    ),
    ThreadMessage(
      id="msg_1v3fWutSoKb0jpb90yL5FoP1",
      assistant_id=None,
      content=[
        MessageContentText(
          text=Text(
            annotations=[],
            value="Which lesson in the Intro to Generative AI class explains token cost?",
          ),
          type="text",
        )
      ],
      created_at=1712772653,

```

```

        file_ids=[],
        metadata={},
        object="thread.message",
        role="user",
        run_id=None,
        thread_id="thread_pSXweFuKtYQA3hcpJs9ashQp",
    ),
],
object="list",
first_id="msg_DI45Bp0aNdaByyoWMIpj3pot",
last_id="msg_1v3fWutSoKb0jpb90yL5FoP1",
has_more=False,
)

```

Yes, this is big and intimidating, but when broken down it's actually pretty simple. Each message in this response is encapsulated in a `ThreadMessage()` object. You can see that this response contains two messages: the initial message from the user, and the assistant's response. The most recent message—in this case the assistant's response—is always the first object within the response. This means that in most cases, you can reference the latest message that you want to display to the user with `thread_messages.data[0].content[0].text.value`.

RUNS

A run is how an updated thread is executed. An execution is when the assistant uses the model, any specified tools, and the context of the thread message history to generate a response. This is how a run is created:

```

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

```

`threads.runs.create()` has two required parameters, `thread_id` and `assistant_id`. You can probably guess what they are! `thread_id` is the ID of the thread to be run, and `assistant_id` is the ID of the assistant that should execute the run.

`threads.runs.create()` returns an object that looks like this:


```

Run(
  id="run_G3saOqF0lgxkGzKcFdio5ocU",
  assistant_id="asst_mKqxycEu1VvAhCm9ljAewEN6",
  cancelled_at=None,
  completed_at=None,
  created_at=1712765431,
  expires_at=1712766031,
  failed_at=None,
  file_ids=[],
  instructions="You are a study partner for students who are newer to technology. When
you answer prompts, do so with simple language suitable for someone learning
fundamental concepts.",
  last_error=None,
  metadata={},
  model="gpt-3.5-turbo",
  object="thread.run",
  required_action=None,
  started_at=None,
  status="queued",
  thread_id="thread_yMHG8Ca2UpvBITxUO2uvl3NC",
  tools=[],
  usage=None,
  temperature=1.0,
)

```

This response has a lot of information in it. The most important property to notice right now is `status`. It takes a bit of time for the OpenAI server (whichever US-based server your assistant may be using) to process the request and generate a response. You cannot display the response message to the user until the request is processed and the response is returned, so `status` is a way for you to monitor the status of the run.

Possible values for `status` are: `queued`, `in_progress`, `requires_action`, `cancelling`, `cancelled`, `failed`, `completed`, or `expired`.

To check the status of a run, you can use `threads.runs.retrieve()`:

```
run = client.beta.threads.runs.retrieve(  
    thread_id = thread.id,  
    run_id = run.id  
)
```

`threads.runs.retrieve()` has two required parameters, `thread_id` and `run_id`. `thread_id` is the ID of the thread you want to retrieve run information for, and `run_id` is the ID of the run.

`threads.runs.retrieve()` returns an object that looks like this:

```
Run(  
    id="run_G3saOqF0lgxkGzKcFdio5ocU",  
    assistant_id="asst_mKqxycEu1VvAhCm9ljAewEN6",  
    cancelled_at=None,  
    completed_at=None,  
    created_at=1712771330,  
    expires_at=1712771930,  
    failed_at=None,  
    file_ids=[],  
    instructions="You are a study partner for students who are newer to technology. When  
you answer prompts, do so with simple language suitable for someone learning  
fundamental concepts.",  
    last_error=None,  
    metadata={},  
    model="gpt-3.5-turbo",  
    object="thread.run",  
    required_action=None,  
    started_at=1712771331,  
    status="in_progress",  
    thread_id="thread_yMHG8Ca2UpvBITxUO2uvl3NC",  
    tools=[],  
    usage=None,  
    temperature=1.0,  
)
```

As you can see, the properties in this object are identical to those in the object returned by `threads.runs.create()`. However, the value of `status` has changed from “queued” to

“in_progress”. This value can be checked until it is “complete”, at which point the response will be available and you can display it to the user.

You’ll see all of these in action in the rest of this lesson. Let’s gooooo, Skillcrusher! ➡➡➡

7.4 How to Build an Assistant

So far in this lesson you’ve learned what the Assistants API is, what assistants are, the four objects needed for working with assistants, and how to work with those objects.

Phew, that’s a lot! Good job, Skillcrusher 🏆

Now it’s time for you to learn how to put all of that knowledge together in order to build an assistant that will serve as a study partner for this class.

That’s right! The study buddy you’re about to start building can be used to support your learning in this class, helping you search for concepts you learned throughout the lessons, have concepts explained in different ways, find supplemental learning materials, and more!

First things first, let’s look at psuedocode representing the logic for your new assistant:

create the assistant

create a thread

prompt the user to input a message

use the prompt to create a message within the thread

create a run

monitor the run status

extract the most recent message content when the run is completed

display the message to the user

The basic structure of this is similar to the chatbot you created using the Chat Completions API: receive a prompt from the user, process the prompt, and display the response back to the user. The big difference is the logic in the middle that is specific to the Assistants API.

Next, let's map the Assistants API objects and API calls needed for each of these steps. You should look back to the previous lesson step as needed, but hopefully these will look somewhat familiar by now!

create the assistant

```
assistant = client.beta.assistants.create(  
    name = "",  
    model = "",  
    instructions = ""  
)
```

create a thread

```
thread = client.beta.threads.create()
```

prompt the user to input a message

```
user_input = input("You: ")
```

use the prompt to create a message within the thread

```
message = client.beta.threads.messages.create(  
    thread_id = thread.id,  
    role = "",  
    content = user_input  
)
```

create a run

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

monitor the run status

```
run = client.beta.threads.runs.retrieve(  
    thread_id = thread.id,
```

```

        run_id = run.id
    )

    extract the most recent message content when the run is completed
    if run.status == "completed":
        thread_messages = client.beta.threads.messages.list(
            thread_id = thread.id
        )

        message_for_user = thread_messages.data[0].content[0].text.value
    )

    display the message to the user
    print("Assistant: " + message_for_user)

```

In the next challenge you'll bring this all together with a bit more code to create your study buddy. Woohoo!

7.5 Start Building Your Assistant

It's time to roll your sleeves alllllll the way up and get to work! In this challenge, you'll take the code snippets that we mapped to pseudocode in the last step and start building your study buddy. This study buddy won't be able to answer questions about this class curriculum just yet—that functionality will be added in the next lesson! It will however be able to answer general questions about web development, Python, and artificial intelligence.

Here's what you'll do:

- Create the assistant
- Create a thread
- Prompt the user for a message
- Add the message to the thread
- Create a run
- Monitor the run status until it is complete
- When the response is available, display it to the user

We won't be showing you every single line of code to write in the sections below so that you can get some valuable practice writing the script mostly on your own. Refer back to the previous steps in this lesson for help as needed.

First, create a new directory for your assistant by navigating to your home directory (or wherever you'd like to create your new project directory) and in the command line run `mkdir assistant`. Then `cd` into the assistant directory and create a new file called `assistant.py` using the `touch` command. Finally, run `source openai-env/bin/activate` to create your virtual environment.

CREATE THE ASSISTANT

Start your file as you did in your `chatbot.py` script by importing the OpenAI module and create an instance of it.

Then, create your assistant using `client.beta.assistants.create()`. Specify the model, which is the only required parameter. Give your assistant a name and instructions as well—while these are not required parameters, it's best practice to include them. Also add a `tools` parameter with an empty list.

CREATE A THREAD

Next you need to create a thread. For this you should use `client.beta.threads.create()`, which does not have any required parameters.

PROMPT THE USER FOR A MESSAGE

Prompt the user to submit a message using `input()`.

ADD THE MESSAGE TO THE THREAD

Now that you have a message from the user, you can add it to the thread. Use `client.beta.threads.messages.create()`, and remember that it needs the thread ID, the role (which in this case is "user"), and the content which is the user's input.

CREATE A RUN

Next create a run using `client.beta.threads.runs.create()`. This needs both the thread ID and assistant ID.

MONITOR THE RUN STATUS

So far we've followed the pseudocode in the previous lesson step. Before you retrieve the thread's run, you need a bit of logic to monitor the status of the chat, at regular intervals, until the status of the run is complete. Only then will you extract the message. To add this logic, you can use a `while` loop.

First, at the top of your file, import the `time` module. You'll use the time module to add a one-second delay between each run status check.

Head back down to the bottom of your file and start your `while` loop:

```
while True:
```

Within the while loop, add a one-second delay using `time.sleep()` :

```
while True:
    time.sleep(1)
```

Note: you can change the one-second delay to two or even three seconds if you'd like. Developer's choice!

Next, call `threads.runs.retrieve()`, which needs the thread ID and run ID:

```
while True:
    time.sleep(1)
    run = client.beta.threads.runs.retrieve(
        thread_id = thread.id,
        run_id = run.id
    )
```

Finally, check the run status, and if it has completed, break out of the while loop:

```
while True:
    time.sleep(1)
    run = client.beta.threads.runs.retrieve(
        thread_id = thread.id,
        run_id = run.id
    )
```

```
if run.status == "completed":  
    break
```

Now you know that the run has successfully completed and the thread has been updated with the latest message—the assistant's response.

DISPLAY THE RESPONSE TO THE USER

Now that you know that the thread has been updated with the message generated during the run execution, you can get all the message threads using `threads.messages.list()` and the `thread_id`.

Remember, all messages are listed in reverse order, so you need to extract the value of the first message from the messages list. This line can feel tricky to write. If you're unsure, you can review the messages section in this lesson step.

Finally, display the message back to the user using `print()`.

Time to run your script. As a reminder, you can run your script the same way you did when you built your chatbot:

On a Mac:

```
python3 assistant.py
```

On Windows:

```
python assistant.py
```

When you run your script, it should behave like a single-turn conversation with the user adding input and the assistant responding. If it works: woohoo! If it doesn't: go through the steps again, and refer back to the pseudocode in the previous lesson step. You got this!

When you're all done, deactivate the virtual environment. Next, make a new GitHub repository for this project, add and commit your work, and push up to GitHub. Enter the link to your new GitHub repository below

If you would like to check the formatting of your code, take a look at the solution code.

7.6 Zhuzh Up Your Study Buddy Assistant

You're going to jump right into your next challenge, which is to zhuzh up your study buddy assistant. This challenge will require you to revisit concepts you've learned throughout this class and apply them to your new assistant. Here's what you'll do:

- Make your assistant multi-turn
- Add more messaging for the user
- Add logging

MAKE YOUR ASSISTANT MULTI-TURN

In order to make your assistant multi-turn, you'll use a while loop just as you did to make your chatbot multi-turn. However, this version is more complex due to the fact that you *also* need to create runs, monitor the status of each run, and handle the outcome of each run within the while loop. Let's break it down step-by-step.

First, create a new function toward the top of your file called `process_run()`. This function will have two parameters, `thread_id` and `assistant_id`:

```
def process_run(thread_id, assistant_id):
```

Next, move all of the run logic into this new function:

```
def process_run(thread_id, assistant_id):
    new_run = client.beta.threads.runs.create(
        thread_id = thread_id,
        assistant_id = assistant_id
    )

    while True:
        time.sleep(1)
        run_check = client.beta.threads.runs.retrieve(
            thread_id = thread_id,
            run_id = new_run.id
        )
        if run_check.status == "completed":
            break
```

Make sure you update anything that references the thread ID and the assistant ID to use `thread_id` and `assistant_id`, the parameter names in the function declaration.

Now you need to rework the while loop to return the run object when it is finished processing.

Notice I said finished processing and not completed. This function is only responsible for returning the run object when it is finished processing, regardless of the outcome. Let's review all the possible values for the run status:

`queued`, `in_progress`, `requires_action`, `cancelling`, `cancelled`, `failed`, `completed`, `expired`

Now let's group these into two lists: one for statuses that mean the run has finished processing, and one for statuses that means the run is still processing.

Finished processing: `cancelled`, `failed`, `completed`, `expired`

Still processing: `queued`, `in_progress`, `requires_action`, `cancelling`

Return the run object when the run status is any of the "finished processing" values:

```
def process_run(thread_id, assistant_id):
    new_run = client.beta.threads.runs.create(
        thread_id = thread_id,
        assistant_id = assistant_id
    )

    while True:
        time.sleep(1)
        run_check = client.beta.threads.runs.retrieve(
            thread_id = thread_id,
            run_id = new_run.id
        )
        if run_check.status in ["cancelled", "failed", "completed", "expired"]:
            return run_check
```

Now your while loop will continue until the `run_check` status is any of the finished processing statuses, at which point it will return the run object.

Now, directly below the line of code where you create the thread, create your new while loop:

```
thread = client.beta.threads.create()
```

```
while True:
```

Within this while loop, you need to:

- Prompt the user for input
- If the user types "exit", break out of the while loop
- Create a message
- Process the run
- Display a message back to the user

First, prompt the user for input and break out of the while loop if they type "exit":

```
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break
```

Next, create a message with the user's input. You have this code, you just need to move it into the while loop and make sure `content` is set to `user_input`:

```
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    message = client.beta.threads.messages.create(
        thread_id = thread.id,
        role = "user",
        content = user_input
```

```
)
```

Next, create the run by calling your new `process_run()` function:

```
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    message = client.beta.threads.messages.create(
        thread_id = thread.id,
        role = "user",
        content = user_input
    )

    run = process_run(thread.id, assistant.id)
```

Almost done! Now you need to handle the object returned by the `process_run()` function. The way you do this depends on the status of the run.

If the status is “completed”, retrieve the list of messages and display the latest message value to the user:

```
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    message = client.beta.threads.messages.create(
        thread_id = thread.id,
        role = "user",
        content = user_input
    )

    run = process_run(thread.id, assistant.id)

    if run.status == "completed":
```

```
thread_messages = client.beta.threads.messages.list(
    thread_id = thread.id
)
print("\nAssistant: " + thread_messages.data[0].content[0].text.value + "\n")
```

If the status is any of "cancelled", "failed", or "expired", tell the user that an error occurred and that they should try again:

```
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    message = client.beta.threads.messages.create(
        thread_id = thread.id,
        role = "user",
        content = user_input
    )

    run = process_run(thread.id, assistant.id)

    if run.status == "completed":
        thread_messages = client.beta.threads.messages.list(
            thread_id = thread.id
        )
        print("\nAssistant: " + thread_messages.data[0].content[0].text.value + "\n")

    if run.status in ["cancelled", "failed", "expired"]:
        print("\nAssistant: An error has occurred, please try again.\n")
```

And that's it! Go ahead and test out your updated script.

ADD MORE MESSAGING FOR THE USER

Users feel more confident and excited to use your application if they understand how the application works and what actions are taking place as they use it. A great way to achieve this is to give the user explicit notices and feedback. So let's do that!

First, greet your user by adding a prompt at the beginning of your while loop asking for their name. You did this in your chatbot, so refer back to that challenge if you need help.

Next, let your user know that the script is processing their prompt after they enter one. This is important feedback for the user because it's possible for a run to take several seconds to process, and the user will feel more confident knowing that something is taking place.

Head up to your `process_run()` function, and below the `time.sleep(1)` line within the while loop, print a message:

```
while True:
    time.sleep(1)
    print("Thinking...")
```

Save your file and try it out.

This is great, but we can make it more fun!!

Above the while loop, create a list of short phrases that are appropriate feedback for the user:

```
phrases = ["Thinking", "Pondering", "Dotting the i's", "Achieving world peace"]
```

As you can see, you can have a little fun with it 😊

Now, update your print line to print a random selection from the `phrases` list. For this you need to use the built-in Python module `random`, so import it at the top of your file. Then update your print line to use `random.choice()`, which selects a random item from a list. Don't forget to add an ellipsis at the end!

```
print(random.choice(phrases) + "...")
```

Save your file and try it out. Do you agree that this provides a better user experience?

ADD LOGGING

Remember back in lesson 6 when you logged your chatbot's token usage? You can take a similar approach here to log errors that occur while a user is interacting with your assistant. We won't walk through this step-by-step, but here are some hints to guide you:

- Import the `logging` and `datetime` modules at the top of your file.
- Configure your log by using `logging.getLogger()` and `logging.basicConfig()`.
- Create a function called `log_run()` that accepts the run status as an argument and logs an error if the status is cancelled, failed, or expired
- Call the `log_run()` function toward the bottom of your file, below `process_run()`.

OTHER OPTIONAL THINGS TO TRY

Here are a few other suggestions for how you can zhuzh up your assistant. It's not required that you complete these, but if you're interested, we've included them in the solution code.

- Add a goodbye message for the user—including their name—if they type exit.
- Create a function that accepts the run status as an argument and returns an appropriate response based on the status value.
 - This function would be called directly below the `process_run()` function call.
 - Can the responses be more specific based on the status? Do they need to be?

When you're satisfied with the functionality of your assistant and you've tested it, add and commit your work, push to your GitHub repo, and enter the link below.