ChatterBot: Build a Chatbot With Python

Table of Contents

Chatbots can provide real-time customer support and are therefore a valuable asset in many industries. When you understand the basics of the **ChatterBot** library, you can **build and train** a **self-learning chatbot** with just a few lines of Python code.

You'll get the basic chatbot up and running right away in step one, but the most interesting part is the learning phase, when you get to train your chatbot. The quality and preparation of your training data will make a big difference in your chatbot's performance.

To simulate a real-world process that you might go through to create an industry-relevant chatbot, you'll learn how to customize the chatbot's responses. You'll do this by preparing **WhatsApp chat data** to train the chatbot. You can apply a similar process to train your bot from different conversational data in any domain-specific topic.

In this tutorial, you'll learn how to:

- Build a **command-line chatbot** with ChatterBot
- **Train** the chatbot to customize its responses
- **Export** your WhatsApp chat history
- Perform **data cleaning** on the chat export using **regular expressions**
- **Retrain** the chatbot with industry-specific data

You'll also learn how ChatterBot stores your training data, and you'll find suggestions and pointers for next steps, so you can start collecting real user data and let the chatbot learn from it.

Overall, in this tutorial, you'll quickly run through the basics of creating a chatbot with ChatterBot and learn how Python allows you to get fun and useful results without needing to write a lot of code.

Step 1: Create a Chatbot Using Python ChatterBot

In this step, you'll set up a virtual environment and install the necessary dependencies. You'll also create a working command-line chatbot that can reply to you—but it won't have very interesting replies for you yet.

To get started with your chatbot project, create and activate a virtual environment, then install `chatterbot` and `pytz`:

```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install chatterbot==1.0.4 pytz
```

Running these commands in your terminal application installs ChatterBot and its dependencies into a new Python virtual environment.

After the installation is complete, running `python -m pip freeze` should bring up list of installed dependencies that's similar to what you can find in the provided sample code's `requirements.txt` file:

```
# bot.py

from chatterbot import ChatBot

chatbot = ChatBot("Chatpot")

  exit_conditions = (":q", "quit", "exit")
  while True:
  query = input("> ")
  if query in exit_conditions:
  break
  else:
  print(f"🪴 {chatbot.get_response(query)}")
```

After importing `ChatBot` in line 3, you create an instance of `ChatBot` in line 5. The only required argument is a name, and you call this one `"Chatpot"`. No, that's not a typo—you'll actually build a chatty flowerpot chatbot in this tutorial! You'll soon notice that pots may not be the best conversation partners after all.

In line 8, you create a `while` loop that'll keep looping unless you enter one of the exit conditions defined in line 7. Finally, in line 13, you call `.get_response()` on the `ChatBot` instance that you created earlier and pass it the user input that you collected in line 9 and assigned to `query`.

The call to `.get_response()` in the final line of the short script is the only interaction with your `chatbot`. And yet—you have a functioning command-line chatbot that you can take for a spin.

When you run `bot.py`, ChatterBot might download some data and language models associated with the NLTK project. It'll print some information about that to your console. Python won't download this data again during subsequent runs.

If you're ready to communicate with your freshly homegrown `Chatpot`, then you can go ahead and run the Python file:

Even if your chat-pot doesn't have much to say yet, it's already learning and growing. To test this out, stop the current session. You can do this by typing one of the exit conditions—`":q"`, `"quit"`, or `"exit"`. Then start the chatbot another time. Enter a different message, and you'll notice that the chatbot remembers what you typed during the previous run:

Step 2: Begin Training Your Chatbot

In the previous step, you built a chatbot that you could interact with from your command line. The chatbot started from a clean slate and wasn't very interesting to talk to.

In this step, you'll train your chatbot using `ListTrainer` to make it a little smarter from the start. You'll also learn about built-in trainers that come with ChatterBot, including their limitations.

Your chatbot doesn't have to start from scratch, and ChatterBot provides you with a quick way to train your bot. You'll use ChatterBot's `ListTrainer` to provide some conversation samples that'll give your chatbot more room to grow:

```
# bot.py

from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer

chatbot = ChatBot("Chatpot")

trainer = ListTrainer(chatbot)
trainer.train([
    "Hi",
    "Welcome, friend 🤗",
])
trainer.train([
    "Are you a plant?",
    "No, I'm the pot below the plant!",
])

exit_conditions = (":q", "quit", "exit")
while True:
    query = input("> ")
    if query in exit_conditions:
        break
    else:
        print(f"🤗 {chatbot.get_response(query)}")
```

You can run more than one training session, so in lines 13 to 16, you add another statement and another reply to your chatbot's database.

If you now run the interactive chatbot once again using `python bot.py`, you can elicit somewhat different responses from it than before:

```
> hi
🤗 Welcome, friend 🤗
> hello
🤗 are you a plant?
> me?
🤗 are you a plant?
> yes
```

```
 hi
> are you a plant?
 No, I'm the pot below the plant!
> cool
 Welcome, friend 
```

The conversation isn't yet fluent enough that you'd like to go on a second date, but there's additional context that you didn't have before! When you train your chatbot with more data, it'll get better at responding to user inputs.

The ChatterBot library comes with some corpora that you can use to train your chatbot. However, at the time of writing, there are some issues if you try to use these resources straight out of the box.

Step 3: Export a WhatsApp Chat

.

To export the history of a conversation that you've had on WhatsApp, you need to open the conversation on your phone. Once you're on the conversation screen, you can access the export menu:

1. Click on the three dots (⋮) in the top right corner to open the main menu.
2. Choose *More* to bring up additional menu options.
3. Select *Export chat* to create a TXT export of your conversation.

Step 4: Clean Your Chat Export

In this step, you'll clean the WhatsApp chat export data so that you can use it as input to train your chatbot on an industry-specific topic. In this example, the topic will be … houseplants!

Most data that you'll use to train your chatbot will require some kind of cleaning before it can produce useful results. It's just like the old saying goes:

Garbage in, garbage out (Source)

Take some time to explore the data that you're working with and to identify potential issues:

```
9/15/22, 14:50 - Messages and calls are end-to-end encrypted.
↪No one outside of this chat, not even WhatsApp, can read
↪or listen to them. Tap to learn more.

...

9/15/22, 14:50 - Philipp: I'm ready to talk about plants!
```

```
...

9/16/22, 06:34 - Martin: <Media omitted>

...
```

For example, you may notice that the first line of the provided chat export isn't part of the conversation. Also, each actual message starts with metadata that includes a date, a time, and the username of the message sender.

If you scroll further down the conversation file, you'll find lines that aren't real messages. Because you didn't include media files in the chat export, WhatsApp replaced these files with the text `<Media omitted>`.

All of this data would interfere with the output of your chatbot and would certainly make it sound much less conversational. Therefore, it's a good idea to remove this data.

Open up a new Python file to preprocess your data before handing it to ChatterBot for training. Start by reading in the file content and removing the chat metadata:

```
1# cleaner.py
2
3import re
4
5def remove_chat_metadata(chat_export_file):
6    date_time = r"(\d+\/\d+\/\d+,\s\d+:\d+)"  # e.g. "9/16/22, 06:34"
7    dash_whitespace = r"\s-\s"  # " - "
8    username = r"([\w\s]+)"  # e.g. "Martin"
9    metadata_end = r":\s"  # ": "
10    pattern = date_time + dash_whitespace + username + metadata_end
11
12    with open(chat_export_file, "r") as corpus_file:
13        content = corpus_file.read()
14    cleaned_corpus = re.sub(pattern, "", content)
15    return tuple(cleaned_corpus.split("\n"))
16
17if __name__ == "__main__":
18    print(remove_chat_metadata("chat.txt"))
```

This function removes conversation-irrelevant message metadata from the chat export file using the built-in `re` module, which allows you to work with regular expressions:

- **Line 3** imports `re`.
- **Lines 6 to 9** define multiple regex patterns. Constructing multiple patterns helps you keep track of what you're matching and gives you the flexibility to use the separate capturing groups to apply further preprocessing later on.

For example, with access to `username`, you could chunk conversations by merging messages sent consecutively by the same user.

- **Line 10** concatenates the regex patterns that you defined in lines 6 to 9 into a single pattern. The complete pattern matches all the metadata that you want to remove.
- **Lines 12 and 13** open the chat export file and read the data into memory.
- **Line 14** uses `re.sub()` to replace each occurrence of the pattern that you defined in `pattern` with an empty string (`""`), effectively deleting it from the string.
- **Line 15** first splits the file content string into list items using `.split("\n")`. This breaks up `cleaned_corpus` into a list where each line represents a separate item. Then, you convert this list into a tuple and return it from `remove_chat_metadata()`.
- **Lines 17 and 18** use Python's name-main idiom to call `remove_chat_metadata()` with `"chat.txt"` as its argument, so that you can inspect the output when you run the script.

Eventually, you'll use `cleaner` as a module and import the functionality directly into `bot.py`. But while you're developing the script, it's helpful to inspect intermediate outputs, for example with a `print()` call, as shown in line 18.