CREATE CHATBOT IN PYTHON

Tensorflow:

Demo

At the end of this tutorial, you'll have a command-line chatbot that can respond to your inputs with semi-meaningful replies:

You'll achieve that by preparing WhatsApp chat data and using it to train the chatbot. Beyond learning from your automated training, the chatbot will improve over time as it gets more exposure to questions and replies from user interactions.

Project Overview

The ChatterBot library combines language corpora, text processing, machine learning algorithms, and data storage and retrieval to allow you to build flexible chatbots.

You can build an industry-specific chatbot by training it with relevant data. Additionally, the chatbot will remember user responses and continue

building its internal graph structure to improve the responses that it can give.

In this tutorial, you'll start with an untrained chatbot that'll showcase how quickly you can create an interactive chatbot using Python's ChatterBot. You'll also notice how small the vocabulary of an untrained chatbot is.

Next, you'll learn how you can train such a chatbot and check on the slightly improved results. The more plentiful and high-quality your training data is, the better your chatbot's responses will be.

Therefore, you'll either fetch the conversation history of one of your WhatsApp chats or use the provided chat.txt file that you can download here:

It's rare that input data comes exactly in the form that you need it, so you'll clean the chat export data to get it into a useful input format. This process will show you some tools you can use for data cleaning, which may help you prepare other input data to feed to your chatbot.

After data cleaning, you'll retrain your chatbot and give it another spin to experience the improved performance.

When you work through this process from start to finish, you'll get a good idea of how you can build and train a Python chatbot with the ChatterBot library so that it can provide an interactive experience with relevant replies.

Prerequisites

Before you get started, make sure that you have a Python version available that works for this ChatterBot project. What version of Python you need depends on your operating system:

- Windows
- Linux
- macOS

You need to use a Python version below 3.8 to successfully work with the recommended version of ChatterBot in this tutorial. You can install Python 3.7.9 using pyenv-win.

If you've installed the right Python version for your operating system, then you're ready to get started. You'll touch on a handful of Python concepts while working through the tutorial:

- Conditional statements
- while loops for iteration
- Lists and tuples
- Python functions
- Substring checks and substring replacement
- File input/output
- Python comprehensions and generator expressions
- Regular expressions (regex) using re

If you're comfortable with these concepts, then you'll probably be comfortable writing the code for this tutorial. If you don't have all of the prerequisite knowledge before starting this tutorial, that's okay! In fact, you might learn more by going ahead and getting started. You can always stop and review the resources linked here if you get stuck.

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:

- Windows
- Linux + macOS

```
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:

With the installation out of the way, and ignoring some of the issues that the library currently has, you're ready to get started! Create a new Python file, call it bot.py, and add the code that you need to get a basic chatbot up and running:

```
1# bot.py
2
3from chatterbot import ChatBot
4
5chatbot = ChatBot("Chatpot")
6
7exit_conditions = (":q", "quit", "exit")
8while True:
9    query = input("> ")
10    if query in exit_conditions:
11        break
12    else:
13        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:

```
$ python bot.py
```

After the language models are set up, you'll see the greater than sign (>) that you defined in bot.py as your input prompt. You can now start to interact with your chatty pot:

```
> hello
 hello
> are you a plant?
 hello
> can you chat, pot?
 hello
```

Well … your chat-pot is *responding*, but it's really struggling to branch out. Tough to expect more from a potted plant—after all, it's never gotten to see the world!

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:

```
> hi
 hello
> what's up?
```

```
□ are you a plant?
```

During the first run, ChatterBot created a SQLite database file where it stored all your inputs and connected them with possible responses. There should be three new files that have popped up in your working directory:

```
./
├── bot.py
├── db.sqlite3
├── db.sqlite3-shm
└── db.sqlite3-wal
```

ChatterBot uses the default SQLStorageAdapter and creates a SQLite file database unless you specify a different storage adapter.

Because you said both *hello* and *hi* at the beginning of the chat, your chat-pot learned that it can use these messages interchangeably. That means if you chat a lot with your new chatbot, it'll gradually have better replies for you. But improving its responses manually sounds like a long process!

Now that you've created a working command-line chatbot, you'll learn how to train it so you can have slightly more interesting conversations.

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:

```
1# bot.py
```

```
2
3from chatterbot import ChatBot
4from chatterbot.trainers import ListTrainer
5
6chatbot = ChatBot("Chatpot")
7
8trainer = ListTrainer(chatbot)
9trainer.train([
10    "Hi",
11    "Welcome, friend 🤗",
12])
13trainer.train([
14    "Are you a plant?",
15    "No, I'm the pot below the plant!",
16])
17
18exit_conditions = (":q", "quit", "exit")
19while True:
20    query = input("> ")
21    if query in exit_conditions:
22        break
23    else:
24        print(f"🪴 {chatbot.get_response(query)}")
```

In line 4, you import ListTrainer, to which you pass your chatbot on line 8 to create trainer.

In lines 9 to 12, you set up the first training round, where you pass a list of two strings to trainer.train(). Using .train() injects entries into your database to build upon the graph structure that ChatterBot uses to choose possible replies.

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.

While the provided corpora might be enough for you, in this tutorial you'll skip them entirely and instead learn how to adapt your own conversational input data for training with ChatterBot's ListTrainer.

To train your chatbot to respond to industry-relevant questions, you'll probably need to work with custom data, for example from existing support requests or chat logs from your company.

Moving forward, you'll work through the steps of converting chat data from a WhatsApp conversation into a format that you can use to train your chatbot. If your own resource is WhatsApp conversation data, then you can

use these steps directly. If your data comes from elsewhere, then you can adapt the steps to fit your specific text format.

To start off, you'll learn how to export data from a WhatsApp chat conversation.

Step 3: Export a WhatsApp Chat

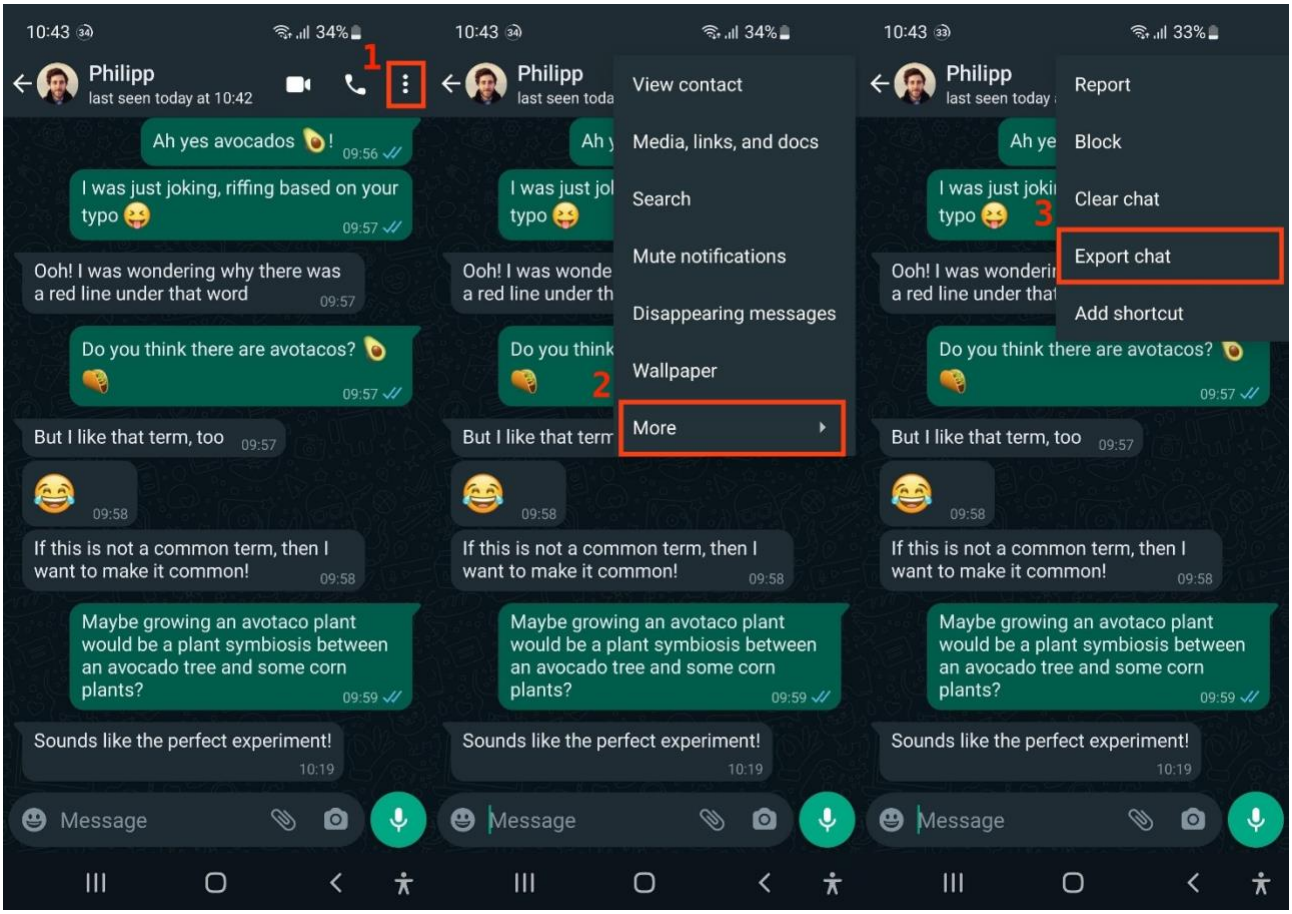
At the end of this step, you'll have downloaded a TXT file that contains the chat history of a WhatsApp conversation. If you don't have a WhatsApp account or don't want to work with your own conversational data, then you can download a sample chat export below:

If you're going to work with the provided chat history sample, you can skip to the next section, where you'll clean your chat export.

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:

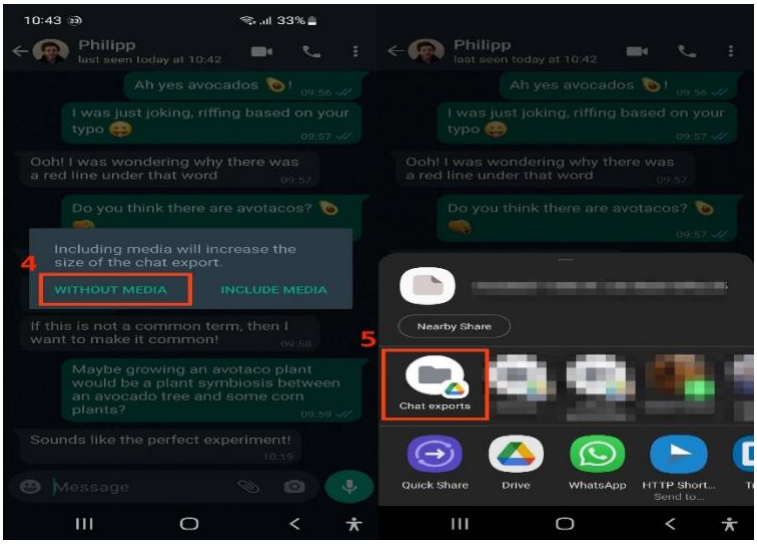
1. Click on the three dots (⋮) in the top right corner to open the main menu.
2. Choose *More* to bring up additional menu options.
3. Select *Export chat* to create a TXT export of your conversation.

In the stitched-together screenshots below, you can see the three consecutive steps numbered and outlined in red:



Once you've clicked on *Export chat*, you need to decide whether or not to include media, such as photos or audio messages. Because your chatbot is only dealing with text, select *WITHOUT MEDIA*. Then, you can declare where you'd like to send the file.

Again, you can see an example of these next steps in two stitched-together WhatsApp screenshots with red numbers and outlines below:



In this example, you saved the chat export file to a Google Drive folder named *Chat exports*. You'll have to set up that folder in your Google Drive before you can select it as an option. Of course, you don't need to use Google Drive. As long as you save or send your chat export file so that you can access to it on your computer, you're good to go.

Once that's done, switch back to your computer. Find the file that you saved, and download it to your machine.

Specifically, you should save the file to the folder that also contains bot.py and rename it chat.txt. Then, open it with your favorite text editor to inspect the data that you received:

```
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, 14:49 - Philipp: Hi Martin, Philipp here!
9/15/22, 14:50 - Philipp: I'm ready to talk about plants!
9/15/22, 14:51 - Martin: Oh that's great!
9/15/22, 14:52 - Martin: I've been waiting for a good convo about
↪ plants for a long time
9/15/22, 14:52 - Philipp: We all have.
9/15/22, 14:52 - Martin: Did you know they need water to grow?
...
```

If you remember how ChatterBot handles training data, then you'll see that the format isn't ideal to use for training.

ChatterBot uses complete lines as messages when a chatbot replies to a user message. In the case of this chat export, it would therefore include all the message metadata. That means your friendly pot would be studying the dates, times, and usernames! Not exactly great conversation fertilizer.

To avoid this problem, you'll clean the chat export data before using it to train your chatbot.

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:

```
# cleaner.py

import re

def remove_chat_metadata(chat_export_file):
    date_time = r"(\d+\/\d+\/\d+,\s\d+:\d+)"  # e.g. "9/16/22, 06:34"
    dash_whitespace = r"\s-\s"  # " - "
    username = r"([\w\s]+)"  # e.g. "Martin"
    metadata_end = r":\s"  # ": "
    pattern = date_time + dash_whitespace + username + metadata_end

    with open(chat_export_file, "r") as corpus_file:
        content = corpus_file.read()
    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.

After removing the message metadata from each line, you also want to remove a few complete lines that aren't relevant for the conversation. To do this, create a second function in your data cleaning script:

```
1# cleaner.py
2
3# ...
4
5def remove_non_message_text(export_text_lines):
6    messages = export_text_lines[1:-1]
7
8    filter_out_msgs = ("<Media omitted>",)
9    return tuple((msg for msg in messages if msg not in filter_out_msgs))
10
11if __name__ == "__main__":
12    message_corpus = remove_chat_metadata("chat.txt")
13    cleaned_corpus = remove_non_message_text(message_corpus)
14    print(cleaned_corpus)
```

In remove_non_message_text(), you've written code that allows you to remove irrelevant lines from the conversation corpus:

- **Line 6** removes the first introduction line, which every WhatsApp chat export comes with, as well as the empty line at the end of the file.
- **Line 8** creates a tuple where you can define what strings you want to exclude from the data that'll make it to training. For now, it only contains one string, but if you wanted to remove other content as well, you could quickly add more strings to this tuple as items.
- **Line 9** filters messages for the strings defined in filter_out_msgs using a generator expression that you convert to a tuple before returning it.

Finally, you've also changed lines 12 to 14. You now collect the return value of the first function call in the variable message_corpus, then use it as an argument to remove_non_message_text(). You save the result of that function call to cleaned_corpus and print that value to your console on line 14.

Because you want to treat cleaner as a module and run the cleaning code in bot.py, it's best to now refactor the code in the name-main idiom into a main function that you can then import and call in bot.py:

```
# cleaner.py

import re

def clean_corpus(chat_export_file):
    message_corpus = remove_chat_metadata(chat_export_file)
    cleaned_corpus = remove_non_message_text(message_corpus)
    return cleaned_corpus

# ...

# Deleted: if __name__ == "__main__":
```

You refactor your code by moving the function calls from the name-main idiom into a dedicated function, clean_corpus(), that you define toward the top of the file. In line 6, you replace "chat.txt" with the parameter chat_export_file to make it more general. You'll provide the filename when calling the function. The clean_corpus() function returns the cleaned corpus, which you can use to train your chatbot.

After creating your cleaning module, you can now head back over to bot.py and integrate the code into your pipeline.

Step 5: Train Your Chatbot on Custom Data and Start Chatting

In this step, you'll train your chatbot with the WhatsApp conversation data that you cleaned in the previous step. You'll end up with a chatbot that you've trained on industry-specific conversational data, and you'll be able to chat with the bot—about houseplants!

Open up bot.py and include calls to your cleaning functions in the code:

```
1# bot.py
2
3from chatterbot import ChatBot
4from chatterbot.trainers import ListTrainer
5from cleaner import clean_corpus
6
7CORPUS_FILE = "chat.txt"
8
9chatbot = ChatBot("Chatpot")
10
11trainer = ListTrainer(chatbot)
12cleaned_corpus = clean_corpus(CORPUS_FILE)
13trainer.train(cleaned_corpus)
14
15exit_conditions = (":q", "quit", "exit")
16while True:
17    query = input("> ")
18    if query in exit_conditions:
19        break
20    else:
21        print(f"🤖 {chatbot.get_response(query)}")
```

You don't need to do a lot of extra work in this file, because you've prepared your cleaning functions well:

- **Line 5** imports clean_corpus() from cleaner.
- **Line 7** defines the filename of your chat export file. In this example, you assume that it's called "chat.txt", and it's located in the same directory as bot.py. If you need more advanced path handling, then take a look at Python's pathlib module.
- **Line 12** applies your cleaning code to the chat history file and returns a tuple of cleaned messages, which you call cleaned_corpus.
- **Line 13** finally uses that data as input to .train(), effectively training your chatbot with the WhatsApp conversation data.

It's important to know that ChatterBot will treat an iterable with more than two items slightly differently than when there are only two items, like you had in your previous training runs:

> If you provide longer lists of training conversations, then this will establish each item in the list as a possible response to its predecessor in the list. (Source)

Depending on your input data, this may or may not be exactly what you want. For the provided WhatsApp chat export data, this isn't ideal because not every line represents a question followed by an answer.

To deal with this, you could apply additional preprocessing on your data, where you might want to group all messages sent by the same person into one line, or chunk the chat export by time and date. That way, messages sent within a certain time period could be considered a single conversation.

There's always more to clean to improve your input data! For this tutorial, however, it's time to give the conversational houseplant chatbot another go. Run bot.py and chat with your chatty pot to test how its responses have changed due to training:

```
> hi
 Welcome, friend 😂
> thanks for the green welcome
 I let you
> you let me be here?
 It's a monsters!
> did you mean monstera?
 The leafs are getting dryer and dryer. But she's growing new ones
> who?
 Do raindrops touch their leaves?
> very philosophical!
 Lol
> ;)
```

 I don't grow any crop at home
> no crops in pots
 Ah, gotcha!
> ah, I just thought you meant that
 But water makes sense
> yes, water in pots
 Do you have any plant care pro tips?
> water.
 I always thought that love and music was more than enough
> haha
 Well, no. Just the balcony. This worked best
> ok, balcony is notes
 I let you

Your chatbot has increased its range of responses based on the training data that you fed to it. As you might notice when you interact with your chatbot, the responses don't always make a lot of sense.

ChatterBot uses complete sentences as replies, and you trained it on just a small sample of conversational data. The small size of the training data means that it doesn't have a big pool of replies to pick from. And because the data is conversational, not all of the sentences are very useful as replies. But Chatpot is doing all it can to find the best matching reply to any new message that you type!

To select a response to your input, ChatterBot uses the BestMatch logic adapter by default. This logic adapter uses the Levenshtein distance to compare the input string to all statements in the database. It then picks a reply to the statement that's closest to the input string.

If you use well-structured input data, then the default settings of ChatterBot give you decent results out of the box. And if you're ready to do some extra work to get just what you want, then you're in luck! ChatterBot allows for a lot of customization and provides some instructions to guide you in the right direction:

Topic	Approach	Instructions
Training	Inherit from Trainer	Create a new training class
Input preprocessing	Write a function that takes and returns a Statement	Create a new preprocessor
Selecting responses	Inherit from LogicAdapter	Create a new logic adapter
Storing data	Inherit from StorageAdapter	Create a new storage adapter
Comparing statements	Write a function that takes two statements and returns a number between 0 and 1	Create a new comparison function

ChatterBot provides you with reasonable defaults. But if you want to customize any part of the process, then it gives you all the freedom to do so.

In this section, you put everything back together and trained your chatbot with the cleaned corpus from your WhatsApp conversation chat export. At this point, you can already have fun conversations with your chatbot, even though they may be somewhat nonsensical. Depending on the amount and quality of your training data, your chatbot might already be more or less useful.

Conclusion

Congratulations, you've built a Python chatbot using the ChatterBot library! Your chatbot isn't a smarty plant just yet, but everyone has to start

somewhere. You already helped it grow by training the chatbot with preprocessed conversation data from a WhatsApp chat export.

In this tutorial, you learned how to:

- Build a **command-line chatbot** with ChatterBot
- **Train** a 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

Because the industry-specific chat data in the provided WhatsApp chat export focused on houseplants, Chatpot now has some opinions on houseplant care. It'll readily share them with you if you ask about it—or really, when you ask about *anything*.

With big data comes big results! You can imagine that training your chatbot with more input data, particularly more relevant data, will produce better results.

Next Steps

ChatterBot provides a way to install the library as a Django app. As a next step, you could integrate ChatterBot in your Django project and deploy it as a web app.

You can also swap out the database back end by using a different storage adapter and connect your Django ChatterBot to a production-ready database.

After you've completed that setup, your deployed chatbot can keep improving based on submitted user responses from all over the world.

Even if you keep running your chatbot on the CLI for now, there are many ways that you can improve the project and continue to learn about the ChatterBot library:

- **Handle more edge cases:** Your regex pattern might not catch all WhatsApp usernames. You can throw some edge cases at it and improve the stability of your parsing while building tests for your code.
- **Improve conversations:** Group your input data as conversations so that your training input considers consecutive messages sent by the same user within an hour a single message.
- **Parse the ChatterBot corpus:** Skip the dependency conflicts, install PyYAML directly, and parse some of the training corpora provided in chatterbot-corpus yourself. Use one or more of them to continue training your chatbot.
- **Build a custom preprocessor:** ChatterBot can modify user input before sending it to a logic adapter. You can use built-in preprocessors, for example to remove whitespace. Build a custom preprocessor that can replace swear words in your user input.
- **Include additional logic adapters:** ChatterBot comes with a few preinstalled logic adapters, such as ones for mathematical evaluations and time logic. Add these logic adapters to your chatbot so it can perform calculations and tell you the current time.
- **Write a custom logic adapter:** Create a custom logic adapter that triggers on specific user inputs, for example when your users ask for a joke.
- **Incorporate an API call:** Build a logic adapter that can interact with an API service, for example by repurposing your weather CLI project so that it works within your chatbot.

There's a lot that you can do! Check out what your chatbot suggests:

```
> what should i do next?
 Yeah! I want them to be strong and take care of themselves at some point
```

Great advice! Or … at least it *seems* like your chatbot is telling you that you should help it become more self-sufficient?

A great next step for your chatbot to become better at handling inputs is to include more and better training data. If you do that, and utilize all the

features for customization that ChatterBot offers, then you can create a chatbot that responds a little more on point than Chatpot here.

If you're not interested in houseplants, then pick your own chatbot idea with unique data to use for training. Repeat the process that you learned in this tutorial, but clean and use your own data for training.

Did you decide to adapt your chatbot for a specific use case? Did you manage to have a philosophical conversation with it? Or did your chatbot keep switching topics in a funny way? Share your experience in the comments below!

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

!

Project:

Name : Muralidharan. P

Dept : CSE 3rd year

Reg no : 621421104037

College code :6214

Group: IBM-GROUP 5