ChatterBot Documentation

Release 0.7.6

Gunther Cox

Sep 19, 2017

Contents

ChatterBot is a Python library that makes it easy to generate automated responses to a user's input. ChatterBot uses a selection of machine learning algorithms to produce different types of responses. This makes it easy for developers to create chat bots and automate conversations with users. For more details about the ideas and concepts behind ChatterBot see the *process flow diagram*.

An example of typical input would be something like this:

```
user: Good morning! How are you doing?
bot:  I am doing very well, thank you for asking.
user: You're welcome.
bot:  Do you like hats?
```

CHAPTER 1

Language Independence

The language independent design of ChatterBot allows it to be trained to speak any language. Additionally, the machine-learning nature of ChatterBot allows an agent instance to improve it's own knowledge of possible responses as it interacts with humans and other sources of informative data.

CHAPTER 2

How ChatterBot Works

ChatterBot is a Python library designed to make it easy to create software that can engage in conversation.

An *untrained instance* of ChatterBot starts off with no knowledge of how to communicate. Each time a user enters a *statement*, the library saves the text that they entered and the text that the statement was in response to. As ChatterBot receives more input the number of responses that it can reply and the accuracy of each response in relation to the input statement increase.

The program selects the closest matching *response* by searching for the closest matching known statement that matches the input, it then chooses a response from the selection of known responses to that statement.

CHAPTER 3

Process flow diagram

CHAPTER 4

Contents:

Installation

The recommended method for installing ChatterBot is by using pip.

Installing from PyPi

If you are just getting started with ChatterBot, it is recommended that you start by installing the latest version from the Python Package Index (PyPi). To install ChatterBot from PyPi using pip run the following command in your terminal.

```
pip install chatterbot
```

Installing from GitHub

You can install the latest **development** version of ChatterBot directly from GitHub using `pip`.

```
pip install git+git://github.com/gunthercox/ChatterBot.git@master
```

Installing from source

1. Download a copy of the code from GitHub. You may need to install git.

```
git clone https://github.com/gunthercox/ChatterBot.git
```

2. Install the code you have just downloaded using pip

```
pip install ./ChatterBot
```

Checking the version of ChatterBot that you have installed

If you already have ChatterBot installed and you want to check what version you have installed you can run the following command.

```
python -m chatterbot --version
```

Upgrading ChatterBot to the latest version

Upgrading to Newer Releases

Like any software, changes will be made to ChatterBot over time. Most of these changes are improvements. Frequently, you don't have to change anything in your code to benifit from a new release.

Occasionally there are changes that will require modifications in your code or there will be changes that make it possible for you to improve your code by taking advantage of new features.

To view a record of ChatterBot's history of changes, visit the releases tab on ChatterBot's GitHub page.

- https://github.com/gunthercox/ChatterBot/releases

Use the pip command to upgrade your existing ChatterBot installation by providing the –upgrade parameter:

```
pip install chatterbot --upgrade
```

Quick Start Guide

Create a new chat bot

Note: If you are using Python 2.7, be sure that the unicode header is the first line of your Python file: `# -*- coding: utf-8 -*-`

```
from chatterbot import ChatBot
chatbot = ChatBot("Ron Obvious")
```

Note: The only required parameter for the *ChatBot* is a name. This can be anything you want.

Training your ChatBot

After creating a new ChatterBot instance it is also possible to train the bot. Training is a good way to ensure that the bot starts off with knowledge about specific responses. The current training method takes a list of statements that represent a conversation. Additional notes on training can be found in the *Training* documentation.

Note: Training is not required but it is recommended.

```
from chatterbot.trainers import ListTrainer

conversation = [
    "Hello",
    "Hi there!",
    "How are you doing?",
    "I'm doing great.",
    "That is good to hear",
    "Thank you.",
    "You're welcome."
]

chatbot.set_trainer(ListTrainer)
chatbot.train(conversation)
```

Get a response

```
response = chatbot.get_response("Good morning!")
print(response)
```

Read only mode

Your ChatterBot will learn based on each new input statement it receives. If you want to disable this learning feature after your bot has been trained, you can set *read_only=True* as a parameter when initializing the bot.

```
chatbot = ChatBot("Johnny Five", read_only=True)
```

ChatterBot Tutorial

This tutorial will guide you through the process of creating a simple command-line chat bot using ChatterBot.

Getting help

If you're having trouble with this tutorial, you can post a message on Gitter to chat with other ChatterBot users who might be able to help.

If you believe that you have encountered an error in ChatterBot, please open a ticket on GitHub: https://github.com/gunthercox/ChatterBot/issues/new

Creating your first chat bot

Create a new file named *chatbot.py*. Then open *chatbot.py* in your editor of choice.

Before we do anything else, ChatterBot needs to be imported. The import for ChatterBot should look like the following line.

```
from chatterbot import ChatBot
```

Create a new instance of the `ChatBot` class.

```
bot = ChatBot('Norman')
```

This line of code has created a new chat bot named *Norman*. There is a few more parameters that we will want to specify before we run our program for the first time.

Setting the storage adapter

ChatterBot comes with built in adapter classes that allow it to connect to different types of databases. In this tutorial, we will be using the `SQLStorageAdapter` which allows the chat bot to connect to SQL databases. By default, this adapter will create a SQLite database.

The `database` parameter is used to specify the path to the database that the chat bot will use. For this example we will call the database *database.sqlite3*. this file will be created automatically if it doesn't already exist.

```
bot = ChatBot(
    'Norman',
    storage_adapter='chatterbot.storage.SQLStorageAdapter',
    database='./database.sqlite3'
)
```

Note: The SQLStorageAdapter is ChatterBot's default adapter. If you do not specify an adapter in your constructor, the SQLStorageAdapter adapter will be used automatically.

Input and output adapters

Next, we will add in parameters to specify the input and output terminal adapter. The input terminal adapter simply reads the user's input from the terminal. The output terminal adapter prints the chat bot's response.

```
bot = ChatBot(
    'Norman',
    storage_adapter='chatterbot.storage.SQLStorageAdapter',
    input_adapter='chatterbot.input.TerminalAdapter',
    output_adapter='chatterbot.output.TerminalAdapter',
    database='./database.sqlite3'
)
```

Specifying logic adapters

The *logic_adapters* parameter is a list of logic adapters. In ChatterBot, a logic adapter is a class that takes an input statement and returns a response to that statement.

You can choose to use as many logic adapters as you would like. In this example we will use two logic adapters. The TimeLogicAdapter returns the current time when the input statement asks for it. The MathematicalEvaluation adapter solves math problems that use basic operations.

```
bot = ChatBot(
    'Norman',
    storage_adapter='chatterbot.storage.SQLStorageAdapter',
    input_adapter='chatterbot.input.TerminalAdapter',
    output_adapter='chatterbot.output.TerminalAdapter',
    logic_adapters=[
        'chatterbot.logic.MathematicalEvaluation',
```

```
        'chatterbot.logic.TimeLogicAdapter'
    ],
    database='./database.sqlite3'
)
```

Getting a response from your chat bot

Next, you will want to create a while loop for your chat bot to run in. By breaking out of the loop when specific exceptions are triggered, we can exit the loop and stop the program when a user enters *ctrl+c*.

```
while True:
    try:
     bot_input = bot.get_response(None)

    except(KeyboardInterrupt, EOFError, SystemExit):
        break
```

Training your chat bot

At this point your chat bot, Norman will learn to communicate as you talk to him. You can speed up this process by training him with examples of existing conversations.

```
bot.train([
    'How are you?',
    'I am good.',
    'That is good to hear.',
    'Thank you',
    'You are welcome.',
])
```

You can run the training process multiple times to reinforce preferred responses to particular input statements. You can also run the train command on a number of different example dialogs to increase the breadth of inputs that your chat bot can respond to.

This concludes this ChatterBot tutorial. Please see other sections of the documentation for more details and examples.

Up next: *Examples*

Examples

The following examples are available to help you get started with ChatterBot.

Simple Example

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot

# Create a new chat bot named Charlie
chatbot = ChatBot(
```

```
    'Charlie',
    trainer='chatterbot.trainers.ListTrainer'
)

chatbot.train([
    "Hi, can I help you?",
    "Sure, I'd to book a flight to Iceland.",
    "Your flight has been booked."
])

# Get a response to the input text 'How are you?'
response = chatbot.get_response('I would like to book a flight.')

print(response)
```

Terminal Example

This example program shows how to create a simple terminal client that allows you to communicate with your chat bot by typing into your terminal.

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot


# Uncomment the following lines to enable verbose logging
# import logging
# logging.basicConfig(level=logging.INFO)

# Create a new instance of a ChatBot
bot = ChatBot(
    "Terminal",
    storage_adapter="chatterbot.storage.SQLStorageAdapter",
    logic_adapters=[
        "chatterbot.logic.MathematicalEvaluation",
        "chatterbot.logic.TimeLogicAdapter",
        "chatterbot.logic.BestMatch"
    ],
    input_adapter="chatterbot.input.TerminalAdapter",
    output_adapter="chatterbot.output.TerminalAdapter",
    database="../database.db"
)

print("Type something to begin...")

# The following loop will execute each time the user enters input
while True:
    try:
        # We pass None to this method because the parameter
        # is not used by the TerminalAdapter
        bot_input = bot.get_response(None)

    # Press ctrl-c or ctrl-d on the keyboard to exit
    except (KeyboardInterrupt, EOFError, SystemExit):
        break
```

Using MongoDB

Before you can use ChatterBot's built in adapter for MongoDB, you will need to install MongoDB. Make sure MongoDB is running in your environment before you execute your program. To tell ChatterBot to use this adapter, you will need to set the *storage_adapter* parameter.

```
storage_adapter="chatterbot.storage.MongoDatabaseAdapter"
```

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot


# Uncomment the following lines to enable verbose logging
# import logging
# logging.basicConfig(level=logging.INFO)

# Create a new ChatBot instance
bot = ChatBot(
    'Terminal',
    storage_adapter='chatterbot.storage.MongoDatabaseAdapter',
    logic_adapters=[
        'chatterbot.logic.BestMatch'
    ],
    filters=[
        'chatterbot.filters.RepetitiveResponseFilter'
    ],
    input_adapter='chatterbot.input.TerminalAdapter',
    output_adapter='chatterbot.output.TerminalAdapter',
    database='chatterbot-database'
)

print('Type something to begin...')

while True:
    try:
        bot_input = bot.get_response(None)

    # Press ctrl-c or ctrl-d on the keyboard to exit
    except (KeyboardInterrupt, EOFError, SystemExit):
        break
```

Time and Mathematics Example

ChatterBot has natural language evaluation capabilities that allow it to process and evaluate mathematical and time-based inputs.

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot


bot = ChatBot(
    "Math & Time Bot",
    logic_adapters=[
        "chatterbot.logic.MathematicalEvaluation",
        "chatterbot.logic.TimeLogicAdapter"
    ],
```

```
    input_adapter="chatterbot.input.VariableInputTypeAdapter",
    output_adapter="chatterbot.output.OutputAdapter"
)

# Print an example of getting one math based response
response = bot.get_response("What is 4 + 9?")
print(response)

# Print an example of getting one time based response
response = bot.get_response("What time is it?")
print(response)
```

Gitter Example

ChatterBot works great with chat rooms. An example for the popular service *Gitter* demonstrates this.

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot
from settings import GITTER


# Uncomment the following lines to enable verbose logging
# import logging
# logging.basicConfig(level=logging.INFO)


chatbot = ChatBot(
    'GitterBot',
    gitter_room=GITTER['ROOM'],
    gitter_api_token=GITTER['API_TOKEN'],
    gitter_only_respond_to_mentions=False,
    input_adapter='chatterbot.input.Gitter',
    output_adapter='chatterbot.output.Gitter',
    trainer='chatterbot.trainers.ChatterBotCorpusTrainer'
)

chatbot.train('chatterbot.corpus.english')

# The following loop will execute each time the user enters input
while True:
    try:
        response = chatbot.get_response(None)

    # Press ctrl-c or ctrl-d on the keyboard to exit
    except (KeyboardInterrupt, EOFError, SystemExit):
        break
```

Using SQL Adapter

ChatterBot data can be saved and retrieved from SQL databases.

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot

# Uncomment the following lines to enable verbose logging
```

```
# import logging
# logging.basicConfig(level=logging.INFO)

# Create a new instance of a ChatBot
bot = ChatBot(
    "SQLMemoryTerminal",
    storage_adapter='chatterbot.storage.SQLStorageAdapter',
    logic_adapters=[
        "chatterbot.logic.MathematicalEvaluation",
        "chatterbot.logic.TimeLogicAdapter",
        "chatterbot.logic.BestMatch"
    ],
    input_adapter="chatterbot.input.TerminalAdapter",
    output_adapter="chatterbot.output.TerminalAdapter",
)

print("Type something to begin...")

# The following loop will execute each time the user enters input
while True:
    try:
        # We pass None to this method because the parameter
        # is not used by the TerminalAdapter
        bot_input = bot.get_response(None)

    # Press ctrl-c or ctrl-d on the keyboard to exit
    except (KeyboardInterrupt, EOFError, SystemExit):
        break
```

More Examples

Even more examples can be found in the *examples* directory in on GitHub: https://github.com/gunthercox/ChatterBot/tree/master/examples

Training

ChatterBot includes tools that help simplify the process of training a chat bot instance. ChatterBot's training process involves loading example dialog into the chat bot's database. This either creates or builds upon the graph data structure that represents the sets of known statements and responses. When a chat bot trainer is provided with a data set, it creates the necessary entries in the chat bot's knowledge graph so that the statement inputs and responses are correctly represented.

Several training classes come built-in with ChatterBot. These utilities range from allowing you to update the chat bot's database knowledge graph based on a list of statements representing a conversation, to tools that allow you to train your bot based on a corpus of pre-loaded training data.

You can also create your own training class. This is recommended if you wish to train your bot with data you have stored in a format that is not already supported by one of the pre-built classes listed below.

Setting the training class

ChatterBot comes with training classes built in, or you can create your own if needed. To use a training class you must import it and pass it to the *set_trainer()* method before calling *train()*.

Training classes

Training via list data

chatterbot.trainers.**ListTrainer**(*storage*, ***kwargs*)
Allows a chat bot to be trained using a list of strings where the list represents a conversation.

For the training process, you will need to pass in a list of statements where the order of each statement is based on its placement in a given conversation.

For example, if you were to run bot of the following training calls, then the resulting chatterbot would respond to both statements of "Hi there!" and "Greetings!" by saying "Hello".

```
from chatterbot.trainers import ListTrainer

chatterbot = ChatBot("Training Example")
chatterbot.set_trainer(ListTrainer)

chatterbot.train([
    "Hi there!",
    "Hello",
])

chatterbot.train([
    "Greetings!",
    "Hello",
])
```

You can also provide longer lists of training conversations. This will establish each item in the list as a possible response to it's predecessor in the list.

```
chatterbot.train([
    "How are you?",
    "I am good.",
    "That is good to hear.",
    "Thank you",
    "You are welcome.",
])
```

Training with corpus data

chatterbot.trainers.**ChatterBotCorpusTrainer**(*storage*, ***kwargs*)
Allows the chat bot to be trained using data from the ChatterBot dialog corpus.

ChatterBot comes with a corpus data and utility module that makes it easy to quickly train your bot to communicate. To do so, simply specify the corpus data modules you want to use.

```
from chatterbot.trainers import ChatterBotCorpusTrainer

chatterbot = ChatBot("Training Example")
chatterbot.set_trainer(ChatterBotCorpusTrainer)
```

```
chatterbot.train(
    "chatterbot.corpus.english"
)
```

Specifying corpus scope

It is also possible to import individual subsets of ChatterBot's corpus at once. For example, if you only wish to train based on the english greetings and conversations corpora then you would simply specify them.

```
chatterbot.train(
    "chatterbot.corpus.english.greetings",
    "chatterbot.corpus.english.conversations"
)
```

You can also specify file paths to corpus files or directories of corpus files when calling the `train` method.

```
chatterbot.train(
    "./data/greetings_corpus/custom.corpus.json",
    "./data/my_corpus/"
)
```

Training with the Twitter API

`chatterbot.trainers.`**`TwitterTrainer`**(*storage*, ***kwargs*)

Allows the chat bot to be trained using data gathered from Twitter.

Parameters **`random_seed_word`** – The seed word to be used to get random tweets from the Twitter API. This parameter is optional. By default it is the word 'random'.

Create an new app using your twitter account. Once created, it will provide you with the following credentials that are required to work with the Twitter API.

Parameter	Description
`twitter_consumer_key`	Consumer key of twitter app.
`twitter_consumer_secret`	Consumer secret of twitter app.
`twitter_access_token_key`	Access token key of twitter app.
`twitter_access_token_secret`	Access token secret of twitter app.

Twitter training example

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot
from settings import TWITTER
import logging


'''
This example demonstrates how you can train your chat bot
using data from Twitter.

To use this example, create a new file called settings.py.
In settings.py define the following:
```

```
TWITTER = {
    "CONSUMER_KEY": "my-twitter-consumer-key",
    "CONSUMER_SECRET": "my-twitter-consumer-secret",
    "ACCESS_TOKEN": "my-access-token",
    "ACCESS_TOKEN_SECRET": "my-access-token-secret"
}
'''

# Comment out the following line to disable verbose logging
logging.basicConfig(level=logging.INFO)

chatbot = ChatBot(
    "TwitterBot",
    logic_adapters=[
        "chatterbot.logic.BestMatch"
    ],
    input_adapter="chatterbot.input.TerminalAdapter",
    output_adapter="chatterbot.output.TerminalAdapter",
    database="./twitter-database.db",
    twitter_consumer_key=TWITTER["CONSUMER_KEY"],
    twitter_consumer_secret=TWITTER["CONSUMER_SECRET"],
    twitter_access_token_key=TWITTER["ACCESS_TOKEN"],
    twitter_access_token_secret=TWITTER["ACCESS_TOKEN_SECRET"],
    trainer="chatterbot.trainers.TwitterTrainer"
)

chatbot.train()

chatbot.logger.info('Trained database generated successfully!')
```

Training with the Ubuntu dialog corpus

Warning: The Ubuntu dialog corpus is a massive data set. Developers will currently experience significantly decreased performance in the form of delayed training and response times from the chat bot when using this corpus.

chatterbot.trainers.**UbuntuCorpusTrainer**(*storage*, ***kwargs*)

Allow chatbots to be trained with the data from the Ubuntu Dialog Corpus.

This training class makes it possible to train your chat bot using the Ubuntu dialog corpus. Because of the file size of the Ubuntu dialog corpus, the download and training process may take a considerable amount of time.

This training class will handle the process of downloading the compressed corpus file and extracting it. If the file has already been downloaded, it will not be downloaded again. If the file is already extracted, it will not be extracted again.

Creating a new training class

You can create a new trainer to train your chat bot from your own data files. You may choose to do this if you want to train your chat bot from a data source in a format that is not directly supported by ChatterBot.

Your custom trainer should inherit *chatterbot.trainers.Trainer* class. Your trainer will need to have a method named *train*, that can take any parameters you choose.

Take a look at the existing trainer classes on GitHub for examples.

Preprocessors

ChatterBot's *preprocessors* are simple functions that modify the input statement that a chat bot receives before the statement gets processed by the logic adaper.

Here is an example of how to set preprocessors. The `preprocessors` parameter should be a list of strings of the import paths to your preprocessors.

```
chatbot = ChatBot(
    'Bob the Bot',
    preprocessors=[
        'chatterbot.preprocessors.clean_whitespace'
    ]
)
```

Preprocessor functions

ChatterBot comes with several built-in preprocessors.

`chatterbot.preprocessors.`**`clean_whitespace`**(*chatbot*, *statement*)
Remove any consecutive whitespace characters from the statement text.

`chatterbot.preprocessors.`**`unescape_html`**(*chatbot*, *statement*)
Convert escaped html characters into unescaped html characters. For example: "<b>" becomes "<b>".

`chatterbot.preprocessors.`**`convert_to_ascii`**(*chatbot*, *statement*)
Converts unicode characters to ASCII character equivalents. For example: "på fédéral" becomes "pa federal".

Creating new preprocessors

It is simple to create your own preprocessors. A preprocessor is just a function with a few requirements.

1. It must take two parameters, the first is a `ChatBot` instance, the second is a `Statement` instance.
2. It must return a statement instance.

Logic Adapters

Logic adapters determine the logic for how ChatterBot selects a response to a given input statement.

The MultiLogicAdapter

ChatterBot internally uses a special logic adapter that allows it to choose the best response generated by any number of other logic adapters.

Selecting a response from multiple logic adapters

The `MultiLogicAdapter` is used to select a single response from the responses returned by all of the logic adapters that the chat bot has been configured to use. Each response returned by the logic adapters includes a confidence score that indicates the likeliness that the returned statement is a valid response to the input.

Response selection

The `MultiLogicAdapter` will return the response statement that has the greatest confidence score. The only exception to this is a case where multiple logic adapters return the same statement and therefore *agree* on that response.

For this example, consider a scenario where multiple logic adapters are being used. Assume the following results were returned by a chat bot's logic adapters.

Confidence	Statement
0.2	Good morning
0.5	Good morning
0.7	Good night

In this case, two of the logic adapters have generated the same result. When multiple logic adapters come to the same conclusion, that statement is given priority over another response with a possibly higher confidence score. The fact that the multiple adapters agreed on a response is a significant indicator that a particular statement has a greater probability of being a more accurate response to the input.

When multiple adapters agree on a response, the greatest confidence score that was generated for that response will be returned with it.

Methods

class `chatterbot.logic.multi_adapter.`**`MultiLogicAdapter`**(***kwargs*)

MultiLogicAdapter allows ChatterBot to use multiple logic adapters. It has methods that allow ChatterBot to add an adapter, set the chat bot, and process an input statement to get a response.

`add_adapter`(*adapter*, ***kwargs*)

Appends a logic adapter to the list of logic adapters being used.

Parameters **`adapter`** (*LogicAdapter*) – The logic adapter to be added.

`get_adapters`()

Return a list of all logic adapters being used, including system logic adapters.

`get_greatest_confidence`(*statement*, *options*)

Returns the greatest confidence value for a statement that occurs multiple times in the set of options.

Parameters

- **`statement`** – A statement object.
- **`options`** – A tuple in the format of (confidence, statement).

`get_initialization_functions`()

Get the initialization functions for each logic adapter.

`insert_logic_adapter`(*logic_adapter*, *insert_index*, ***kwargs*)

Adds a logic adapter at a specified index.

Parameters

- **`logic_adapter`** (*str*) – The string path to the logic adapter to add.

- **insert_index** (*int*) – The index to insert the logic adapter into the list at.

process(*statement*)
Returns the output of a selection of logic adapters for a given input statement.

Parameters **statement** – The input statement to be processed.

remove_logic_adapter(*adapter_name*)
Removes a logic adapter from the chat bot.

Parameters **adapter_name** (*str*) – The class name of the adapter to remove.

set_chatbot(*chatbot*)
Set the chatbot for each of the contained logic adapters.

How logic adapters select a response

A typical logic adapter designed to return a response to an input statement will use two main steps to do this. The first step involves searching the database for a known statement that matches or closely matches the input statement. Once a match is selected, the second step involves selecting a known response to the selected match. Frequently, there will be a number of existing statements that are responses to the known match.

To help with the selection of the response, several methods are built into ChatterBot for selecting a response from the available options.

Response selection methods

Response selection methods determines which response should be used in the event that multiple responses are generated within a logic adapter.

chatterbot.response_selection.**get_first_response**(*input_statement*, *response_list*)

Parameters

- **input_statement** (Statement) – A statement, that closely matches an input to the chat bot.
- **response_list** (*list*) – A list of statement options to choose a response from.

Returns Return the first statement in the response list.

Return type *Statement*

chatterbot.response_selection.**get_most_frequent_response**(*input_statement*, *response_list*)

Parameters

- **input_statement** (Statement) – A statement, that closely matches an input to the chat bot.
- **response_list** (*list*) – A list of statement options to choose a response from.

Returns The response statement with the greatest number of occurrences.

Return type *Statement*

chatterbot.response_selection.**get_random_response**(*input_statement*, *response_list*)

Parameters

- **input_statement** (Statement) – A statement, that closely matches an input to the chat bot.

- **response_list** (*list*) – A list of statement options to choose a response from.

Returns Choose a random response from the selection.

Return type *Statement*

Use your own response selection method

You can create your own response selection method and use it as long as the function takes two parameters (a statements and a list of statements). The method must return a statement.

```
def select_response(statement, statement_list):

    # Your selection logic

    return selected_statement
```

Setting the response selection method

To set the response selection method for your chat bot, you will need to pass the `response_selection_method` parameter to your chat bot when you initialize it. An example of this is shown below.

```
from chatterbot import ChatBot
from chatterbot.response_selection import get_most_frequent_response

chatbot = ChatBot(
    # ...
    response_selection_method=get_most_frequent_response
)
```

Response selection in logic adapters

When a logic adapter is initialized, the response selection method parameter that was passed to it can be called using `self.select_response` as shown below.

```
response = self.select_response(
    input_statement, list_of_response_options
)
```

Creating a new logic adapter

You can write your own logic adapters by creating a new class that inherits from `LogicAdapter` and overrides the necessary methods established in the `LogicAdapter` base class.

Logic adapter methods

class chatterbot.logic.**LogicAdapter**(***kwargs*)

This is an abstract class that represents the interface that all logic adapters should implement.

can_process(*statement*)

A preliminary check that is called to determine if a logic adapter can process a given statement. By default, this method returns true but it can be overridden in child classes as needed.

Return type bool

class_name
Return the name of the current logic adapter class. This is typically used for logging and debugging.

get_initialization_functions()
Return a dictionary of functions to be run once when the chat bot is instantiated.

process(*statement*)
Override this method and implement your logic for selecting a response to an input statement.

A confidence value and the selected response statement should be returned. The confidence value represents a rating of how accurate the logic adapter expects the selected response to be. Confidence scores are used to select the best response from multiple logic adapters.

The confidence value should be a number between 0 and 1 where 0 is the lowest confidence level and 1 is the highest.

Parameters **statement** (Statement) – An input statement to be processed by the logic adapter.

Return type *Statement*

Example logic adapter

```
from chatterbot.logic import LogicAdapter


class MyLogicAdapter(LogicAdapter):
    def __init__(self, **kwargs):
        super(MyLogicAdapter, self).__init__(**kwargs)

    def can_process(self, statement):
        return True

    def process(self, statement):
        import random

        # Randomly select a confidence between 0 and 1
        confidence = random.uniform(0, 1)

        # For this example, we will just return the input as output
        selected_statement = statement
        selected_statement.confidence = confidence

        return selected_statement
```

Directory structure

If you create your own logic adapter you will need to have it in a separate file from your chat bot. Your directory setup should look something like the following:

```
project_directory
- cool_chatbot.py
- cool_adapter.py
```

Then assuming that you have a class named `MyLogicAdapter` in your *cool_chatbot.py* file, you should be able to specify the following when you initialize your chat bot.

```
ChatBot(
    # ...
    logic_adapters=[
        {
            'import_path': 'cool_chatbot.MyLogicAdapter'
        }
    ]
)
```

Responding to specific input

If you want a particular logic adapter to only respond to a unique type of input, the best way to do this is by overriding the `can_process` method on your own logic adapter.

Here is a simple example. Let's say that we only want this logic adapter to generate a response when the input statement starts with "Hey Mike". This way, statements such as "Hey Mike, what time is it?" will be processed, but statements such as "Do you know what time it is?" will not be processed.

```
def can_process(self, statement):
    if statement.text.startswith('Hey Mike')
        return True
    else:
        return False
```

Interacting with services

In some cases, it is useful to have a logic adapter that can interact with an external service or api in order to complete its task. Here is an example that demonstrates how this could be done. For this example we will use a fictitious API endpoint that returns the current temperature.

```
def can_process(self, statement):
    """
    Return true if the input statement contains
    'what' and 'is' and 'temperature'.
    """
    words = ['what', 'is', 'temperature']
    if all(x in statement.text.split() for x in words)
        return True
    else:
        return False

def process(self, statement):
    from chatterbot.conversation import Statement
    import requests

    # Make a request to the temperature API
    response = requests.get('https://api.temperature.com/current?units=celsius')
    data = response.json()

    # Let's base the confidence value on if the request was successful
    if response.status_code == 200:
        confidence = 1
```

```
    else:
        confidence = 0

    temperature = data.get('temperature', 'unavailable')

    response_statement = Statement('The current temperature is {}'.
↪format(temperature))

    return confidence, response_statement
```

Providing extra arguments

All key word arguments that have been set in your ChatBot class's constructor will also be passed to the `__init__` method of each logic adapter. This allows you to access these variables if you need to use them in your logic adapter. (An API key might be an example of a parameter you would want to access here.)

You can override the `__init__` method on your logic adapter to store additional information passed to it by the ChatBot class.

```
class MyLogicAdapter(LogicAdapter):
    def __init__(self, **kwargs):
        super(MyLogicAdapter, self).__init__(**kwargs)

        self.api_key = kwargs.get('secret_key')
```

The `secret_key` variable would then be passed to the ChatBot class as shown below.

```
chatbot = ChatBot(
    # ...
    secret_key='************************'
 )
```

The logic adapter that your bot uses can be specified by setting the `logic_adapters` parameter to the import path of the logic adapter you want to use.

It is possible to enter any number of logic adapters for your bot to use. If multiple adapters are used, then the bot will return the response with the highest calculated confidence value. If multiple adapters return the same confidence, then the adapter that is entered into the list first will take priority.

```
chatbot = ChatBot(
    "My ChatterBot",
    logic_adapters=[
        "chatterbot.logic.BestMatch"
    ]
)
```

Best Match Adapter

`chatterbot.logic.`**`BestMatch`**`(`*`**kwargs`*`)`

A logic adater that returns a response based on known responses to the closest matches to the input statement.

The `BestMatch` logic adapter selects a response based on the best known match to a given statement.

How it works

The best match adapter uses a function to compare the input statement to known statements. Once it finds the closest match to the input statement, it uses another function to select one of the known responses to that statement.

Setting parameters

```
chatbot = ChatBot(
    "My ChatterBot",
    logic_adapters=[
        {
            "import_path": "chatterbot.logic.BestMatch",
            "statement_comparison_function": "chatterbot.comparisons.levenshtein_
↪distance",
            "response_selection_method": "chatterbot.response_selection.get_first_
↪response"
        }
    ]
)
```

Note: The values for `response_selection_method` and `statement_comparison_function` can be a string of the path to the function, or a callable.

See the *Statement comparison* documentation for the list of functions included with ChatterBot.

See the *Response selection methods* documentation for the list of response selection methods included with ChatterBot.

Time Logic Adapter

`chatterbot.logic.`**`TimeLogicAdapter`**(***kwargs*)

The TimeLogicAdapter returns the current time.

The `TimeLogicAdapter` identifies statements in which a question about the current time is asked. If a matching question is detected, then a response containing the current time is returned.

```
User: What time is it?
Bot: The current time is 4:45PM.
```

Mathematical Evaluation Adapter

`chatterbot.logic.`**`MathematicalEvaluation`**(***kwargs*)

The MathematicalEvaluation logic adapter parses input to determine whether the user is asking a question that requires math to be done. If so, MathematicalEvaluation goes through a set of steps to parse the input and extract the equation that must be solved. The steps, in order, are:

1.Normalize input: Remove punctuation and other irrelevant data

2.Convert words to numbers

3.Extract the equation

4.Simplify the equation

5.Solve the equation & return result

The `MathematicalEvaluation` logic adapter checks a given statement to see if it contains a mathematical expression that can be evaluated. If one exists, then it returns a response containing the result. This adapter is able to handle any combination of word and numeric operators.

```
User: What is four plus four?
Bot: (4 + 4) = 8
```

Low Confidence Response Adapter

This adapter returns a specified default response if a response can not be determined with a high amount of confidence.

`chatterbot.logic.`**`LowConfidenceAdapter`**(***kwargs*)

Returns a default response with a high confidence when a high confidence response is not known.

Low confidence response example

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot


# Create a new instance of a ChatBot
bot = ChatBot(
    'Default Response Example Bot',
    storage_adapter='chatterbot.storage.SQLStorageAdapter',
    logic_adapters=[
        {
            'import_path': 'chatterbot.logic.BestMatch'
        },
        {
            'import_path': 'chatterbot.logic.LowConfidenceAdapter',
            'threshold': 0.65,
            'default_response': 'I am sorry, but I do not understand.'
        }
    ],
    trainer='chatterbot.trainers.ListTrainer'
)

# Train the chat bot with a few responses
bot.train([
    'How can I help you?',
    'I want to create a chat bot',
    'Have you read the documentation?',
    'No, I have not',
    'This should help get you started: http://chatterbot.rtfd.org/en/latest/
↪quickstart.html'
])

# Get a response for some unexpected input
response = bot.get_response('How do I make an omelette?')
print(response)
```

Specific Response Adapter

If the input that the chat bot receives, matches the input text specified for this adapter, the specified response will be returned.

`chatterbot.logic.`**`SpecificResponseAdapter`**(***kwargs*)

Return a specific response to a specific input.

Specific response example

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot


# Create a new instance of a ChatBot
bot = ChatBot(
    'Exact Response Example Bot',
    storage_adapter='chatterbot.storage.SQLStorageAdapter',
    logic_adapters=[
        {
            'import_path': 'chatterbot.logic.BestMatch'
        },
        {
            'import_path': 'chatterbot.logic.SpecificResponseAdapter',
            'input_text': 'Help me!',
            'output_text': 'Ok, here is a link: http://chatterbot.rtfd.org/en/latest/
↪quickstart.html'
        }
    ],
    trainer='chatterbot.trainers.ListTrainer'
)

# Get a response given the specific input
response = bot.get_response('Help me!')
print(response)
```

Input Adapters

ChatterBot's input adapters are designed to allow a chat bot to have a versatile method of receiving or retrieving input from a given source.

Creating a new input adapter

You can write your own input adapters by creating a new class that inherits from `InputAdapter` and overrides the necessary methods established in the base `InputAdapter` class.

`chatterbot.input.`**`InputAdapter`**(***kwargs*)

This is an abstract class that represents the interface that all input adapters should implement.

To create your own input adapter you must override the `process_input` method to return a *Statement* object.

Note that you may need to extend the `__init__` method of your custom input adapter if you intend to save a kwarg parameter that was passed into the chat bot's constructor. (An API key might be an example of a parameter you would want to access here.)

```
from __future__ import unicode_literals
from chatterbot.adapters import Adapter


class InputAdapter(Adapter):
    """
    This is an abstract class that represents the
    interface that all input adapters should implement.
    """

    def process_input(self, *args, **kwargs):
        """
        Returns a statement object based on the input source.
        """
        raise self.AdapterMethodNotImplementedError()

    def process_input_statement(self, *args, **kwargs):
        """
        Return an existing statement object (if one exists).
        """
        input_statement = self.process_input(*args, **kwargs)
        self.logger.info('Recieved input statement: {}'.format(input_statement.text))

        existing_statement = self.chatbot.storage.find(input_statement.text)

        if existing_statement:
            self.logger.info('"{}" is a known statement'.format(input_statement.text))
            input_statement = existing_statement
        else:
            self.logger.info('"{}" is not a known statement'.format(input_statement.
↪text))

        return input_statement
```

The goal of an input adapter is to get input from some source, and then to convert it into a format that ChatterBot can understand. This format is the *Statement* object found in ChatterBot's *conversation* module.

Variable input type adapter

chatterbot.input.**VariableInputTypeAdapter**(***kwargs*)

The variable input type adapter allows the chat bot to accept a number of different input types using the same adapter. This adapter accepts strings, dictionaries and *Statements*.

```
chatbot = ChatBot(
    "My ChatterBot",
    input_adapter="chatterbot.input.VariableInputTypeAdapter"
)
```

Terminal input adapter

chatterbot.input.**TerminalAdapter**(***kwargs*)

A simple adapter that allows ChatterBot to communicate through the terminal.

The input terminal adapter allows a user to type into their terminal to communicate with the chat bot.

```
chatbot = ChatBot(
    "My ChatterBot",
    input_adapter="chatterbot.input.TerminalAdapter"
)
```

Gitter input adapter

chatterbot.input.**Gitter**(***kwargs*)

An input adapter that allows a ChatterBot instance to get input statements from a Gitter room.

```
chatbot = ChatBot(
    "My ChatterBot",
    input_adapter="chatterbot.input.Gitter",
    gitter_api_token="my-gitter-api-token",
    gitter_room="my-room-name",
    gitter_only_respond_to_mentions=True,
)
```

HipChat input adapter

chatterbot.input.**HipChat**(***kwargs*)

An input adapter that allows a ChatterBot instance to get input statements from a HipChat room.

This is an input adapter that allows a ChatterBot instance to communicate through HipChat.

Be sure to also see the documentation for the *HipChat output adapter*.

```
chatbot = ChatBot(
    "My ChatterBot",
    input_adapter="chatterbot.input.HipChat",
    hipchat_host="https://mydomain.hipchat.com",
    hipchat_room="my-room-name",
    hipchat_access_token="my-hipchat-access-token",
)
```

Mailgun input adapter

chatterbot.input.**Mailgun**(***kwargs*)

Get input from Mailgun.

The Mailgun adapter allows a chat bot to receive emails using the Mailgun API.

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot
from settings import MAILGUN

'''
To use this example, create a new file called settings.py.
In settings.py define the following:

MAILGUN = {
    "CONSUMER_KEY": "my-mailgun-api-key",
    "API_ENDPOINT": "https://api.mailgun.net/v3/my-domain.com/messages"
}
```

```
'''

# Change these to match your own email configuration
FROM_EMAIL = "mailgun@salvius.org"
RECIPIENTS = ["gunthercx@gmail.com"]

bot = ChatBot(
    "Mailgun Example Bot",
    mailgun_from_address=FROM_EMAIL,
    mailgun_api_key=MAILGUN["CONSUMER_KEY"],
    mailgun_api_endpoint=MAILGUN["API_ENDPOINT"],
    mailgun_recipients=RECIPIENTS,
    input_adapter="chatterbot.input.Mailgun",
    output_adapter="chatterbot.output.Mailgun",
    storage_adapter="chatterbot.storage.SQLStorageAdapter",
    database="../database.db"
)

# Send an example email to the address provided
response = bot.get_response("How are you?")
print("Check your inbox at ", RECIPIENTS)
```

Microsoft Bot Framework input adapter

chatterbot.input.**Microsoft**(***kwargs*)

An input adapter that allows a ChatterBot instance to get input statements from a Microsoft Bot using *Directline client protocol*. https://docs.botframework.com/en-us/restapi/directline/#navtitle

This is an input adapter that allows a ChatterBot instance to communicate through Microsoft using *direct line client* protocol.

Be sure to also see the documentation for the *Microsoft output adapter*.

```
chatbot = ChatBot(
    "My ChatterBot",
    input_adapter="chatterbot.input.Microsoft",
    directline_host="https://directline.botframework.com",
    directline_conversation_id="IEyJvnDULgn",
    direct_line_token_or_secret="RCurR_XV9ZA.cwA.BKA.
↪iaJrC8xpy8qbOF5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0",
)
```

Output Adapters

Creating a new output adapter

You can write your own output adapters by creating a new class that inherits from `chatterbot.output.OutputAdapter` and overrides the necessary methods established in the `OutputAdapter` class.

To create your own output adapter you must override the `process_response` method to return a *Statement* object.

Note that you may need to extend the `__init__` method of your custom output adapter if you intend to save a kwarg parameter that was passed into the chat bot's constructor. (An API key might be an example of a parameter you would want to access here.)

```
from chatterbot.adapters import Adapter


class OutputAdapter(Adapter):
    """
    A generic class that can be overridden by a subclass to provide extended
    functionality, such as delivering a response to an API endpoint.
    """

    def process_response(self, statement, session_id=None):
        """
        Override this method in a subclass to implement customized functionality.

        :param statement: The statement that the chat bot has produced in response to some input.

        :param session_id: The unique id of the current chat session.

        :returns: The response statement.
        """
        return statement
```

Output format adapter

chatterbot.output.**OutputAdapter**(***kwargs*)

A generic class that can be overridden by a subclass to provide extended functionality, such as delivering a response to an API endpoint.

The output adapter allows the chat bot to return a response in as a *Statement* object.

```
chatbot = ChatBot(
    "My ChatterBot",
    output_adapter="chatterbot.output.OutputAdapter",
    output_format="text"
)
```

Terminal output adapter

chatterbot.output.**TerminalAdapter**(***kwargs*)

A simple adapter that allows ChatterBot to communicate through the terminal.

The output terminal adapter allows a user to type into their terminal to communicate with the chat bot.

```
chatbot = ChatBot(
    "My ChatterBot",
    output_adapter="chatterbot.output.TerminalAdapter"
)
```

Gitter output adapter

chatterbot.output.**Gitter**(***kwargs*)

An output adapter that allows a ChatterBot instance to send responses to a Gitter room.

```
chatbot = ChatBot(
    "My ChatterBot",
    output_adapter="chatterbot.output.Gitter",
    gitter_api_token="my-gitter-api-token",
    gitter_room="my-room-name",
    gitter_only_respond_to_mentions=True,
)
```

HipChat output adapter

chatterbot.output.**HipChat**(***kwargs*)

An output adapter that allows a ChatterBot instance to send responses to a HipChat room.

This is an output adapter that allows a ChatterBot instance to send responses to a HipChat room.

Be sure to also see the documentation for the *HipChat input adapter*.

```
chatbot = ChatBot(
    "My ChatterBot",
    output_adapter="chatterbot.output.HipChat",
    hipchat_host="https://mydomain.hipchat.com",
    hipchat_room="my-room-name",
    hipchat_access_token="my-hipchat-access-token",
)
```

Microsoft Bot Framework output adapter

chatterbot.output.**Microsoft**(***kwargs*)

An output adapter that allows a ChatterBot instance to send responses to a Microsoft bot using *Direct Line client protocol*.

This is an output adapter that allows a ChatterBot instance to send responses to a Microsoft using *Direct Line protocol*.

Be sure to also see the documentation for the *Microsoft input adapter*.

```
chatbot = ChatBot(
    "My ChatterBot",
    output_adapter="chatterbot.output.Microsoft",
    directline_host="https://directline.botframework.com",
    conversation_id="IEyJvnDULgn",
    direct_line_token_or_secret="RCurR_XV9ZA.cwA.BKA.
↪iaJrC8xpy8qbOF5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0",
)
```

Mailgun output adapter

chatterbot.output.**Mailgun**(***kwargs*)

The Mailgun adapter allows the chat bot to send emails using the Mailgun API.

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot
from settings import MAILGUN
```

```
'''
To use this example, create a new file called settings.py.
In settings.py define the following:

MAILGUN = {
    "CONSUMER_KEY": "my-mailgun-api-key",
    "API_ENDPOINT": "https://api.mailgun.net/v3/my-domain.com/messages"
}
'''

# Change these to match your own email configuration
FROM_EMAIL = "mailgun@salvius.org"
RECIPIENTS = ["gunthercx@gmail.com"]

bot = ChatBot(
    "Mailgun Example Bot",
    mailgun_from_address=FROM_EMAIL,
    mailgun_api_key=MAILGUN["CONSUMER_KEY"],
    mailgun_api_endpoint=MAILGUN["API_ENDPOINT"],
    mailgun_recipients=RECIPIENTS,
    input_adapter="chatterbot.input.Mailgun",
    output_adapter="chatterbot.output.Mailgun",
    storage_adapter="chatterbot.storage.SQLStorageAdapter",
    database="../database.db"
)

# Send an example email to the address provided
response = bot.get_response("How are you?")
print("Check your inbox at ", RECIPIENTS)
```

Storage Adapters

Storage adapters provide an interface that allows ChatterBot to connect to different storage backends.

Creating a new storage adapter

You can write your own storage adapters by creating a new class that inherits from `StorageAdapter` and overrides necessary methods established in the base `StorageAdapter` class.

`chatterbot.storage.`**`StorageAdapter`**(*base_query=None*, **args*, ***kwargs*)

This is an abstract class that represents the interface that all storage adapters should implement.

You will then need to implement the interface established by the `StorageAdapter` class.

```
import logging
import os


class StorageAdapter(object):
    """
    This is an abstract class that represents the interface
    that all storage adapters should implement.
    """

    def __init__(self, base_query=None, *args, **kwargs):
```

```
        """
        Initialize common attributes shared by all storage adapters.
        """
        self.kwargs = kwargs
        self.logger = kwargs.get('logger', logging.getLogger(__name__))
        self.adapter_supports_queries = True
        self.base_query = None

    @property
    def Statement(self):
        """
        Create a storage-aware statement.
        """

        if 'DJANGO_SETTINGS_MODULE' in os.environ:
            django_project = __import__(os.environ['DJANGO_SETTINGS_MODULE'])
            if 'use_django_models' in django_project.settings.CHATTERBOT:
                if django_project.settings.CHATTERBOT['use_django_models'] is True:
                    from django.apps import apps
                    Statement = apps.get_model(django_project.settings.CHATTERBOT[
↪'django_app_name'], 'Statement')
                    return Statement

        from chatterbot.conversation.statement import Statement
        statement = Statement
        statement.storage = self
        return statement

    def generate_base_query(self, chatterbot, session_id):
        """
        Create a base query for the storage adapter.
        """
        if self.adapter_supports_queries:
            for filter_instance in chatterbot.filters:
                self.base_query = filter_instance.filter_selection(chatterbot,␣
↪session_id)

    def count(self):
        """
        Return the number of entries in the database.
        """
        raise self.AdapterMethodNotImplementedError(
            'The `count` method is not implemented by this adapter.'
        )

    def find(self, statement_text):
        """
        Returns a object from the database if it exists
        """
        raise self.AdapterMethodNotImplementedError(
            'The `find` method is not implemented by this adapter.'
        )

    def remove(self, statement_text):
        """
        Removes the statement that matches the input text.
        Removes any responses from statements where the response text matches
        the input text.
```

```
        """
        raise self.AdapterMethodNotImplementedError(
            'The `remove` method is not implemented by this adapter.'
        )

    def filter(self, **kwargs):
        """
        Returns a list of objects from the database.
        The kwargs parameter can contain any number
        of attributes. Only objects which contain
        all listed attributes and in which all values
        match for all listed attributes will be returned.
        """
        raise self.AdapterMethodNotImplementedError(
            'The `filter` method is not implemented by this adapter.'
        )

    def update(self, statement):
        """
        Modifies an entry in the database.
        Creates an entry if one does not exist.
        """
        raise self.AdapterMethodNotImplementedError(
            'The `update` method is not implemented by this adapter.'
        )

    def get_random(self):
        """
        Returns a random statement from the database.
        """
        raise self.AdapterMethodNotImplementedError(
            'The `get_random` method is not implemented by this adapter.'
        )

    def drop(self):
        """
        Drop the database attached to a given adapter.
        """
        raise self.AdapterMethodNotImplementedError(
            'The `drop` method is not implemented by this adapter.'
        )

    def get_response_statements(self):
        """
        Return only statements that are in response to another statement.
        A statement must exist which lists the closest matching statement in the
        in_response_to field. Otherwise, the logic adapter may find a closest
        matching statement that does not have a known response.

        This method may be overridden by a child class to provide more a
        efficient method to get these results.
        """
        statement_list = self.filter()

        responses = set()
        to_remove = list()
        for statement in statement_list:
            for response in statement.in_response_to:
```

```
                responses.add(response.text)
        for statement in statement_list:
            if statement.text not in responses:
                to_remove.append(statement)

        for statement in to_remove:
            statement_list.remove(statement)

        return statement_list

    class EmptyDatabaseException(Exception):

        def __init__(self, value='The database currently contains no entries. At␣
↪least one entry is expected. You may need to train your chat bot to populate your␣
↪database.'):
            self.value = value

        def __str__(self):
            return repr(self.value)

    class AdapterMethodNotImplementedError(NotImplementedError):
        """
        An exception to be raised when a storage adapter method has not been␣
↪implemented.
        Typically this indicates that the method should be implement in a subclass.
        """
        pass
```

The storage adapter that your bot uses can be specified by setting the `storage_adapter` parameter to the import path of the storage adapter you want to use.

```
chatbot = ChatBot(
    "My ChatterBot",
    storage_adapter="chatterbot.storage.SQLStorageAdapter"
)
```

SQL Alchemy Storage Adapter

class chatterbot.storage.**SQLStorageAdapter**(***kwargs*)

SQLStorageAdapter allows ChatterBot to store conversation data semi-structutered T-SQL database, virtually, any database that SQL Alchemy supports.

Notes: Tables may change (and will), so, save your training data. There is no data migration (yet). Performance test not done yet. Tests using others databases not finished.

All parameters are optional, by default a sqlite database is used.

It will check if tables is present, if not, it will attempt to create required tables. :keyword database: Used for sqlite database. Ignored if database_uri especified. :type database: str

Parameters

- **database_uri** (*str*) – eg: sqlite:///database_test.db", # use database_uri or database, database_uri can be especified to choose database driver (database parameter will be igored).
- **read_only** (*bool*) – False by default, makes all operations read only, has priority over all DB operations so, create, update, delete will NOT be executed

count()
Return the number of entries in the database.

create()
Populate the database with the tables.

drop()
Drop the database attached to a given adapter.

filter(***kwargs*)
Returns a list of objects from the database. The kwargs parameter can contain any number of attributes. Only objects which contain all listed attributes and in which all values match for all listed attributes will be returned.

find(*statement_text*)
Returns a statement if it exists otherwise None

get_random()
Returns a random statement from the database

remove(*statement_text*)
Removes the statement that matches the input text. Removes any responses from statements where the response text matches the input text.

update(*statement*)
Modifies an entry in the database. Creates an entry if one does not exist.

MongoDB Storage Adapter

class chatterbot.storage.**MongoDatabaseAdapter**(***kwargs*)
The MongoDatabaseAdapter is an interface that allows ChatterBot to store statements in a MongoDB database.

Parameters **database** (*str*) – The name of the database you wish to connect to.

```
database='chatterbot-database'
```

Parameters **database_uri** (*str*) – The URI of a remote instance of MongoDB.

```
database_uri='mongodb://example.com:8100/'
```

deserialize_responses(*response_list*)
Takes the list of response items and returns the list converted to Response objects.

drop()
Remove the database.

filter(***kwargs*)
Returns a list of statements in the database that match the parameters specified.

get_random()
Returns a random statement from the database

get_response_statements()
Return only statements that are in response to another statement. A statement must exist which lists the closest matching statement in the in_response_to field. Otherwise, the logic adapter may find a closest matching statement that does not have a known response.

mongo_to_object(*statement_data*)
Return Statement object when given data returned from Mongo DB.

remove(*statement_text*)
Removes the statement that matches the input text. Removes any responses from statements if the response text matches the input text.

Filters

Filters are an efficient way to create base queries that can be passed to ChatterBot's storage adapters. Filters will reduce the number of statements that a chat bot has to process when it is selecting a response.

How to create a new filter for ChatterBot

This is the basic outline of the code that your filter will need to follow. Each filter should inherit from ChatterBot's *Filter* class and implement a method called *filter_selection*. Everything else that your filter does is up to you.

```
from chatterbot.filters import Filter

class MyFilter(Filter):

    def filter_selection(self, chatterbot):
        # ...
        return query
```

Filter Queries

Filters use a storage adapter's query object to build a query that the adapter can evaluate. The available query methods currently are:

statement_text_equals(statement_text)

This query method returns the current query with the added constraint that the text attribute of any statement returned must be equal to the text specified in the parameter.

statement_text_not_in(statements)

This query method takes a list of statement text values and returns the current query with the added constraint that any statements returned cannot exist in the list specified.

statement_response_list_contains(statement_text)

This query method takes a single statement text value and returns the current query with the added constraint that any statements returned must contain the specified text as a response.

statement_response_list_equals(response_list)

This query method takes a list of statement text values and returns the current query with the added constraint that any statements returned must have an exactly matching list of response values.

Filter Support

Not all storage adapters support filters. If a storage adapter does not support filters, then queries generated by filters will be ignored when using that storage adapter.

A storage adapter only supports filters if it supports querying. You can tell if a storage adapter supports querying by checking if its *adapter_supports_queries* property is set to true.

Setting filters

```
chatbot = ChatBot(
    "My ChatterBot",
    filters=["chatterbot.filters.RepetitiveResponseFilter"]
)
```

Filter classes

class chatterbot.filters.**RepetitiveResponseFilter**
A filter that eliminates possibly repetitive responses to prevent a chat bot from repeating statements that it has recently said.

ChatterBot

The main class `ChatBot` is a connecting point between each of ChatterBot's *adapters*. In this class, an input statement is returned from the *input adapter*, processed and stored by the *logic adapter* and *storage adapter*, and then passed to the output adapter to be returned to the user.

class chatterbot.**ChatBot**(*name*, ***kwargs*)
A conversational dialog chat bot.

Parameters

- **name** (*str*) – A name is the only required parameter for the ChatBot class.
- **storage_adapter** (*str*) – The import path to a storage adapter class.
- **logic_adapters** (*list*) – A list of string paths to each logic adapter the bot uses.
- **input_adapter** (*str*) – The import path to an input adapter class.
- **output_adapter** (*str*) – The import path to an output adapter class.
- **trainer** (*str*) – The import path to the training class to be used with the chat bot.
- **filters** (*list*) – A list of import paths to filter classes to be used by the chat bot.
- **logger** (*logging.Logger*) – A `Logger` object.

classmethod from_config(*config_file_path*)
Create a new ChatBot instance from a JSON config file.

generate_response(*input_statement*, *session_id*)
Return a response based on a given input statement.

get_response(*input_item*, *session_id=None*)
Return the bot's response based on the input.

Parameters **input_item** – An input value.

Returns A response to the input.

Return type *Statement*

initialize()
Do any work that needs to be done before the responses can be returned.

learn_response(*statement*, *previous_statement*)
Learn that the statement provided is a valid response.

set_trainer(*training_class*, ***kwargs*)
Set the module used to train the chatbot.

Parameters

- **training_class** (*Trainer*) – The training class to use for the chat bot.
- ****kwargs** – Any parameters that should be passed to the training class.

train
Proxy method to the chat bot's trainer class.

Example chat bot parameters

```
ChatBot(
    'Northumberland',
    storage_adapter='my.storage.AdapterClass',
    logic_adapters=[
        'my.logic.AdapterClass1',
        'my.logic.AdapterClass2'
    ],
    input_adapter='my.input.AdapterClass',
    output_adapter='my.output.AdapterClass',
    trainer='my.trainer.TrainerClass',
    filters=[
        'my.filter.FilterClass1',
        'my.filter.FilterClass2'
    ],
    logger=custom_logger
)
```

Example expanded chat bot parameters

It is also possible to pass parameters directly to individual adapters. To do this, you must use a dictionary that contains a key called `import_path` which specifies the import path to the adapter class.

```
ChatBot(
    'Leander Jenkins',
    storage_adapter={
        'import_path': 'my.storage.AdapterClass',
        'database_name': 'my-database'
    },
    logic_adapters=[
        {
            'import_path': 'my.logic.AdapterClass1',
            'statement_comparison_function': 'chatterbot.comparisons.levenshtein_
↪distance'
            'response_selection_method': 'chatterbot.response_selection.get_first_
↪response'
```

```
        },
        {
            'import_path': 'my.logic.AdapterClass2',
            'statement_comparison_function': 'my.custom.comparison_function'
            'response_selection_method': 'my.custom.selection_method'
        }
    ],
    input_adapter={
        'import_path': 'my.input.AdapterClass',
        'api_key': '0000-1111-2222-3333-DDDD'
    },
    output_adapter={
        'import_path': 'my.output.AdapterClass',
        'api_key': '0000-1111-2222-3333-DDDD'
    }
)
```

Enable logging

ChatterBot has built in logging. You can enable ChatterBot's logging by setting the logging level in your code.

```
import logging

logging.basicConfig(level=logging.INFO)

ChatBot(
    # ...
)
```

The logging levels available are `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`, and `NOTSET`. See the Python logging documentation for more information.

Using a custom logger

You can choose to use your own custom logging class with your chat bot. This can be useful when testing and debugging your code.

```
import logging

custom_logger = logging.getLogger(__name__)

ChatBot(
    # ...
    logger=custom_logger
)
```

Adapters

ChatterBot uses adapter modules to control the behavior of specific types of tasks. There are four distinct types of adapters that ChatterBot uses, these are storage adapters, input adapters, output adapters and logic adapters.

Adapters types

1. Storage adapters - Provide an interface for ChatterBot to connect to various storage systems such as MongoDB or local file storage.
2. Input adapters - Provide methods that allow ChatterBot to get input from a defined data source.
3. Output adapters - Provide methods that allow ChatterBot to return a response to a defined data source.
4. Logic adapters - Define the logic that ChatterBot uses to respond to input it receives.

Accessing the chatbot instance

When ChatterBot initializes each adapter, it sets an attribute named `chatbot`. The chatbot variable makes it possible for each adapter to have access to all of the other adapters being used. Suppose two input and output adapters need to share some information or perhaps you want to give your logic adapter direct access to the storage adapter. These are just a few cases where this functionality is useful.

Each adapter can be accessed on the chatbot object from within an adapter by referencing *self.chatbot*. Then, `self.chatbot.storage` refers to the storage adapter, `self.chatbot.input` refers to the input adapter, `self.chatbot.output` refers to the current output adapter, and `self.chatbot.logic` refers to the logic adapters.

Adapter defaults

By default, ChatterBot uses the *SQLStorageAdapter* adapter for storage, the *BestMatch* for logic, the *VariableInputTypeAdapter* for input and the *OutputAdapter* for output.

Each adapter can be set by passing in the dot-notated import path to the constructor as shown.

```
bot = ChatBot(
    "Elsie",
    storage_adapter="chatterbot.storage.SQLStorageAdapter",
    input_adapter="chatterbot.input.VariableInputTypeAdapter",
    output_adapter="chatterbot.output.OutputAdapter",
    logic_adapters=[
        "chatterbot.logic.BestMatch"
    ],
)
```

Conversations

Statements

ChatterBot's statement objects represent either an input statement that the chat bot has received from a user, or an output statement that the chat bot has returned based on some input.

class `chatterbot.conversation.`**`Statement`**(*text*, ***kwargs*)

A statement represents a single spoken entity, sentence or phrase that someone can say.

`confidence` = None

ChatterBot's logic adapters assign a confidence score to the statement before it is returned. The confidence score indicates the degree of certainty with which the chat bot believes this is the correct response to the given input.

add_extra_data(*key*, *value*)
This method allows additional data to be stored on the statement object.

Typically this data is something that pertains just to this statement. For example, a value stored here might be the tagged parts of speech for each word in the statement text.

•key = 'pos_tags'

•value = [('Now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('different', 'JJ')]

Parameters

- **key** (*str*) – The key to use in the dictionary of extra data.
- **value** – The value to set for the specified key.

add_response(*response*)
Add the response to the list of statements that this statement is in response to. If the response is already in the list, increment the occurrence count of that response.

Parameters **response** (*Response*) – The response to add.

get_response_count(*statement*)
Find the number of times that the statement has been used as a response to the current statement.

Parameters **statement** (*Statement*) – The statement object to get the count for.

Returns Return the number of times the statement has been used as a response.

Return type int

remove_response(*response_text*)
Removes a response from the statement's response list based on the value of the response text.

Parameters **response_text** (*str*) – The text of the response to be removed.

response_statement_cache
This property is to allow ChatterBot Statement objects to be swappable with Django Statement models.

save()
Save the statement in the database.

serialize()

Returns A dictionary representation of the statement object.

Return type dict

Responses

ChatterBot's response objects represent the relationship between two statements. A response indicates that one statement was issued in response to another statement.

class chatterbot.conversation.**Response**(*text*, ***kwargs*)
A response represents an entity which response to a statement.

Statement-response relationship

ChatterBot stores knowledge of conversations as statements. Each statement can have any number of possible responses.

Each `Statement` object has an `in_response_to` reference which links the statement to a number of other statements that it has been learned to be in response to. The `in_response_to` attribute is essentially a reference to all parent statements of the current statement.

The `Response` object's `occurrence` attribute indicates the number of times that the statement has been given as a response. This makes it possible for the chat bot to determine if a particular response is more commonly used than another.

Statement comparison

ChatterBot uses `Statement` objects to hold information about things that can be said. An important part of how a chat bot selects a response is based on its ability to compare two statements to each other. There are a number of ways to do this, and ChatterBot comes with a handful of methods built in for you to use. This module contains various text-comparison algorithms designed to compare one statement to another.

class `chatterbot.comparisons.`**`JaccardSimilarity`**

Calculates the similarity of two statements based on the Jaccard index.

The Jaccard index is composed of a numerator and denominator. In the numerator, we count the number of items that are shared between the sets. In the denominator, we count the total number of items across both sets. Let's say we define sentences to be equivalent if 50% or more of their tokens are equivalent. Here are two sample sentences:

> The young cat is hungry. The cat is very hungry.

When we parse these sentences to remove stopwords, we end up with the following two sets:

> {young, cat, hungry} {cat, very, hungry}

In our example above, our intersection is {cat, hungry}, which has count of two. The union of the sets is {young, cat, very, hungry}, which has a count of four. Therefore, our Jaccard similarity index is two divided by four, or 50%. Given our similarity threshold above, we would consider this to be a match.

`compare`(*statement*, *other_statement*)

Return the calculated similarity of two statements based on the Jaccard index.

`initialize_nltk_wordnet`()

Download the NLTK wordnet corpora that is required for this algorithm to run only if the corpora has not already been downloaded.

class `chatterbot.comparisons.`**`LevenshteinDistance`**

Compare two statements based on the Levenshtein distance of each statement's text.

For example, there is a 65% similarity between the statements "where is the post office?" and "looking for the post office" based on the Levenshtein distance algorithm.

`compare`(*statement*, *other_statement*)

Compare the two input statements.

Returns The percent of similarity between the text of the statements.

Return type float

class `chatterbot.comparisons.`**`SentimentComparison`**

Calculate the similarity of two statements based on the closeness of the sentiment value calculated for each statement.

`compare`(*statement*, *other_statement*)

Return the similarity of two statements based on their calculated sentiment values.

Returns The percent of similarity between the sentiment value.

Return type float

initialize_nltk_vader_lexicon()
Download the NLTK vader lexicon for sentiment analysis that is required for this algorithm to run.

class chatterbot.comparisons.**SynsetDistance**
Calculate the similarity of two statements. This is based on the total maximum synset similarity between each word in each sentence.

This algorithm uses the wordnet functionality of NLTK to determine the similarity of two statements based on the path similarity between each token of each statement. This is essentially an evaluation of the closeness of synonyms.

compare(*statement*, *other_statement*)
Compare the two input statements.

Returns The percent of similarity between the closest synset distance.

Return type float

initialize_nltk_punkt()
Download required NLTK corpora if they have not already been downloaded.

initialize_nltk_stopwords()
Download required NLTK corpora if they have not already been downloaded.

initialize_nltk_wordnet()
Download required NLTK corpora if they have not already been downloaded.

Use your own comparison function

You can create your own comparison function and use it as long as the function takes two statements as parameters and returns a numeric value between 0 and 1. A 0 should represent the lowest possible similarity and a 1 should represent the highest possible similarity.

```
def comparison_function(statement, other_statement):

    # Your comparison logic

    # Return your calculated value here
    return 0.0
```

Setting the comparison method

To set the statement comparison method for your chat bot, you will need to pass the `statement_comparison_function` parameter to your chat bot when you initialize it. An example of this is shown below.

```
from chatterbot import ChatBot
from chatterbot.comparisons import levenshtein_distance

chatbot = ChatBot(
    # ...
    statement_comparison_function=levenshtein_distance
)
```

Sessions

ChatterBot supports the ability to have multiple concurrent chat sessions. A chat session is where the chat bot interacts with a person, and supporting multiple chat sessions means that your chat bot can have multiple different conversations with different people at the same time.

class `chatterbot.conversation.session.`**`Session`**
A session is an ordered collection of statements that are related to each other.

class `chatterbot.conversation.session.`**`ConversationSessionManager`**
Object to hold and manage multiple chat sessions.

`get`(*session_id*, *default=None*)
Return a session given a unique identifier.

`new`()
Create a new conversation.

`update`(*session_id*, *conversance*)
Add a conversance to a given session if the session exists.

Each session object holds a queue of the most recent communications that have occurred during that session. The queue holds tuples with two values each, the first value is the input that the bot received and the second value is the response that the bot returned.

class `chatterbot.queues.`**`ResponseQueue`**(*maxsize=10*)
An extension of the FixedSizeQueue class with utility methods to help manage the conversation.

`get_last_input_statement`()
Return the last response that was given.

`get_last_response_statement`()
Return the last statement that was received.

Session scope

If two `ChatBot` instances are created, each will have sessions separate from each other.

An adapter can access any session as long as the unique identifier for the session is provided.

Session example

The following example is taken from the Django `ChatterBotView` built into ChatterBot. In this method, the unique identifiers for each chat session are being stored in Django's session objects. This allows different users who interact with the bot through different web browsers to have separate conversations with the chat bot.

```
def post(self, request, *args, **kwargs):
    """
    Return a response to the statement in the posted data.
    """
    input_data = json.loads(request.read().decode('utf-8'))

    self.validate(input_data)

    chat_session = self.get_chat_session(request)

    response = self.chatterbot.get_response(input_data, chat_session.id_string)
```

```
    response_data = response.serialize()

    return JsonResponse(response_data, status=200)
```

Utility Methods

ChatterBot has a utility module that contains a collection of miscellaneous but useful functions.

Module imports

chatterbot.utils.**import_module**(*dotted_path*)
Imports the specified module based on the dot notated import path for the module.

Class initialization

chatterbot.utils.**initialize_class**(*data*, ***kwargs*)

Parameters **data** – A string or dictionary containing a import_path attribute.

Terminal input

chatterbot.utils.**input_function**()
Normalizes reading input between python 2 and 3. The function 'raw_input' becomes 'input' in Python 3.

NLTK corpus downloader

chatterbot.utils.**nltk_download_corpus**(*resource_path*)
Download the specified NLTK corpus file unless it has already been downloaded.

Returns True if the corpus needed to be downloaded.

Stopword removal

chatterbot.utils.**remove_stopwords**(*tokens*, *language*)
Takes a language (i.e. 'english'), and a set of word tokens. Returns the tokenized text with any stopwords removed. Stop words are words like "is, the, a, ..."

Be sure to download the required NLTK corpus before calling this function: - from chatterbot.utils import nltk_download_corpus - nltk_download_corpus('corpora/stopwords')

ChatBot response time

chatterbot.utils.**get_response_time**(*chatbot*)
Returns the amount of time taken for a given chat bot to return a response.

Parameters **chatbot** (ChatBot) – A chat bot instance.

Returns The response time in seconds.

Return type float

Random string generation

chatterbot.utils.**generate_strings**(*total_strings*, *string_length=20*)
Generate a list of random strings.

Parameters

- **total_strings** (*int*) – The number of strings to generate.
- **string_length** (*int*) – The length of each string to generate.

Returns The generated list of random strings.

Return type list

Parsing datetime information

chatterbot.parsing.**datetime_parsing**(*text*, *base_date=datetime.datetime(2017, 9, 19, 7, 57, 44, 237848)*)
Extract datetime objects from a string of text.

ChatterBot Corpus

This is a *corpus* of dialog data that is included in the chatterbot module.

Corpus language availability

Corpus data is user contributed, but it is also not difficult to create one if you are familiar with the language. This is because each corpus is just a sample of various input statements and their responses for the bot to train itself with.

To explore what languages and collections of corpora are available, check out the chatterbot_corpus/data directory in the separate chatterbot-corpus repository.

Note: If you are interested in contributing content to the corpus, please feel free to submit a pull request on ChatterBot's corpus GitHub page. Contributions are welcomed!

https://github.com/gunthercox/chatterbot-corpus

The `chatterbot-corpus` is distributed in its own Python package so that it can be released and upgraded independently from the `chatterbot` package.

Data Format

The data file contained in ChatterBot Corpus is formatted using YAML syntax. This format is used because it is easily readable by both humans and machines.

Table 4.1: Corpus Properties

Property	Required	Description
categories	Required	A list of categories that describe the conversations.
conversations	Optional	A list of conversations. Each conversation is denoted as a list.

Here is an example of the corpus data:

```
categories:
- english
- greetings
conversations:
- - Hello
  - Hi
- - Hello
  - Hi, how are you?
  - I am doing well.
- - Good day to you sir!
  - Why thank you.
- - Hi, How is it going?
  - It's going good, your self?
  - Mighty fine, thank you.
```

The values in this example have the following relationships.

Table 4.2: Evaluated statement relationships

Statement	Response
Hello	Hi
Hello	Hi, how are you?
Hi, how are you?	I am doing well.
Good day to you sir!	Why thank you.
Hi, How is it going?	It's going good, your self?
It's going good, your self?	Mighty fine, thank you.

Exporting your chat bot's database as a training corpus

Now that you have created your chat bot and sent it out into the world, perhaps you are looking for a way to share what it has learned with other chat bots? ChatterBot's training module provides methods that allow you to export the content of your chat bot's database as a training corpus that can be used to train other chat bots.

```
chatbot = ChatBot('Export Example Bot')
chatbot.trainer.export_for_training('./export.yml')
```

Here is an example:

```
# -*- coding: utf-8 -*-
from chatterbot import ChatBot

'''
This is an example showing how to create an export file from
an existing chat bot that can then be used to train other bots.
'''

chatbot = ChatBot(
    'Export Example Bot',
    trainer='chatterbot.trainers.ChatterBotCorpusTrainer'
)

# First, lets train our bot with some data
chatbot.train('chatterbot.corpus.english')
```

```
# Now we can export the data to a file
chatbot.trainer.export_for_training('./my_export.json')
```

Django Integration

ChatterBot has direct support for integration with Django. ChatterBot provides out of the box models and endpoints that allow you build ChatterBot powered Django applications.

Chatterbot Django Settings

You can edit the ChatterBot configuration through your Django settings.py file.

```
CHATTERBOT = {
    'name': 'Tech Support Bot',
    'logic_adapters': [
        'chatterbot.logic.MathematicalEvaluation',
        'chatterbot.logic.TimeLogicAdapter',
        'chatterbot.logic.BestMatch'
    ],
    'trainer': 'chatterbot.trainers.ChatterBotCorpusTrainer',
    'training_data': [
         'chatterbot.corpus.english.greetings'
    ]
}
```

Any setting that gets set in the CHATTERBOT dictionary will be passed to the chat bot that powers your django app.

Additional Django settings

- `use_django_models` [default: True] Use the Django models for storing learned conversation data. If set to False, ChatterBot's non-Django objects will be used.
- `django_app_name` [default: 'django_chatterbot'] The Django app name to look up the models from.

Django Training

Management command

When using ChatterBot with Django, the training process can be executed by running the training management command.

```
python manage.py train
```

Training settings

You can specify any data that you want to be passed to the chat bot trainer in the `training_data` parameter in your `CHATTERBOT` Django settings.

```
CHATTERBOT = {
    # ...
    'trainer': 'chatterbot.trainers.ChatterBotCorpusTrainer',
    'training_data': [
        'chatterbot.corpus.english.greetings'
    ]
}
```

Note: You can also specify paths to corpus files or directories of corpus files in the `training_data` list.

See the documentation for the *Training classes* for other training class options that can be used here.

ChatterBot Django Views

API Views

ChatterBot's django module comes with a pre-built API view that you can make requests against to communicate with your bot from your web application.

The endpoint expects a JSON request with the following data:

```
{"text": "My input statement"}
```

Note: You will need to include ChatterBot's urls in your django url configuration before you can make requests to these views. See the setup instructions for more details.

Webservices

If you want to host your Django app, you need to choose a method through which it will be hosted. There are a few free services that you can use to do this shuch as Heroku and PythonAnyWhere.

WSGI

A common method for serving Python web applications involves using a Web Server Gateway Interface (WSGI) package.

Gunicorn is a great choice for a WSGI server. They have detailed documentation and installation instructions on their website.

Hosting static files

There are numerous ways to host static files for your Django application. One extreemly easy way to do this is by using WhiteNoise, a python package designed to make it possible to serve static files from just about any web application.

Install packages

Install with pip

```
pip install django chatterbot
```

For more details on installing Django, see the Django documentation.

Installed Apps

Add *chatterbot.ext.django_chatterbot* to your *INSTALLED_APPS*

```
INSTALLED_APPS = (
    # ...
    'chatterbot.ext.django_chatterbot',
)
```

API view

If you need a ChatterBot API endpoint you will want to add the following to your urls.py

```
urlpatterns = patterns(
    ...
    url(r'^chatterbot/', include('chatterbot.ext.django_chatterbot.urls', namespace=
↪'chatterbot')),
)
```

Sync your database

```
python manage.py migrate django_chatterbot
```

Note: Looking for a working example? Check our the example Django app using ChatterBot on GitHub: https://github.com/gunthercox/ChatterBot/tree/master/examples/django_app

Frequently Asked Questions

This document is comprised of questions that are frequently asked about ChatterBot and chat bots in general.

Python String Encoding

The Python developer community has published a great article that covers the details of unicode character processing.

- Python 3: https://docs.python.org/3/howto/unicode.html
- Python 2: https://docs.python.org/2/howto/unicode.html

The following notes are intended to help answer some common questions and issues that developers frequently encounter while learning to properly work with different character encodings in Python.

Does ChatterBot handle non-ascii characters?

ChatterBot is able to handle unicode values correctly. You can pass to it non-encoded data and it should be able to process it properly (you will need to make sure that you decode the output that is returned).

Below is one of ChatterBot's tests from tests/test_chatbot.py, this is just a simple check that a unicode response can be processed.

```
def test_get_response_unicode(self):
    """
    Test the case that a unicode string is passed in.
    """
    response = self.chatbot.get_response(u'')
    self.assertGreater(len(response.text), 0)
```

This test passes in both Python 2.7 and 3.x. It also verifies that ChatterBot *can* take unicode input without issue.

How do I fix Python encoding errors?

When working with string type data in Python, it is possible to encounter errors such as the following.

```
UnicodeDecodeError: 'utf8' codec can't decode byte 0x92 in position 48: invalid start␣
↪byte
```

Depending on what your code looks like, there are a few things that you can do to prevent errors like this.

Unicode header

```
# -*- coding: utf-8 -*-
```

When to use the unicode header

If your strings use escaped unicode characters (they look like `u'\u00b0C'`) then you do not need to add the header. If you use strings like `'ØÆÅ'` then you are required to use the header.

If you are using this header it must be the first line in your Python file.

Unicode escape characters

```
>>> print u'\u0420\u043e\u0441\u0441\u0438\u044f'

```

When to use escape characters

Prefix your strings with the unicode escape character `u'...'` when you are using escaped unicode characters.

Import unicode literals from future

```
from __future__ import unicode_literals
```

When to import unicode literals

Use this when you need to make sure that Python 3 code also works in Python 2.

A good article on this can be found here: http://python-future.org/unicode_literals.html

How do I deploy my chat bot to the web?

There are a number of excellent web frameworks for creating Python projects out there. Django and Flask are two excellent examples of these. ChatterBot is designed to be agnostic to the platform it is deployed on and it is very easy to get set up.

To run ChatterBot inside of a web application you just need a way for your application to receive incoming data and to return data. You can do this any way you want, HTTP requests, web sockets, etc.

There are a number of existing examples that show how to do this.

1. An example using Django: https://github.com/gunthercox/ChatterBot/tree/master/examples/django_app
2. An example using Flask: https://github.com/chamkank/flask-chatterbot/blob/master/app.py

Additional details and recommendations for configuring Django can be found in the *Webservices* section of ChatterBot's Django documentation.

Command line tools

ChatterBot comes with a few command line tools that can help

Get the installed ChatterBot version

If have ChatterBot installed and you want to check what version you have then you can run the following command.

```
python -m chatterbot --version
```

Locate NLTK data

ChatterBot uses the Natural Language Toolkit (NLTK) for various language processing functions. ChatterBot downloads additional data that is required by NLTK. The following command can be used to find all NLTK data directories that contain files.

```
python -m chatterbot list_nltk_data
```

Development

As the code for ChatterBot is written, the developers attempt to describe the logic and reasoning for the various decisions that go into creating the internal structure of the software. This internal documentation is intended for future developers and maintaners of the project. A majority of this information is unnecessary for the typical developer using ChatterBot.

It is not always possible for every idea to be documented. As a result, the need may arise to question the developers and maintainers of this project in order to pull concepts from their minds and place them in these documents. Please pull gently.

Contributing to ChatterBot

There are numerous ways to contriubte to ChatterBot. All of which are highly encouraged.

- Contributing bug reports and feature requests
- Contributing documentation
- Contributing code for new features
- Contributing fixes for bugs

Every bit of help received on this project is highly appreciated.

Setting Up a Development Environment

To contribute to ChatterBot's development, you simply need:

- Python
- pip
- A few python packages:

```
pip install requirements.txt
pip install dev-requirements.txt
```

- A text editor

Reporting a Bug

If you discover a bug in ChatterBot and wish to report it, please be sure that you adhere to the following when you report it on GitHub.

1. Before creating a new bug report, please search to see if an open or closed report matching yours already exists.
2. Please include a description that will allow others to recreate the problem you encountered.

Requesting New Features

When requesting a new feature in ChatterBot, please make sure to include the following details in your request.

1. Your use case. Describe what you are doing that requires this new functionality.

Contributing Documentation

ChatterBot's documentation is written in reStructuredText and is compiled by Sphinx. The reStructuredText source of the documentation is located in `docs/`.

To build the documentation yourself, run:

```
sphinx-build ./docs/ ./build/
```

You can then open the index.html file that will be created in the build directory.

Contributing Code

The development of ChatterBot happens on GitHub. Code contributions should be submitted there in the form of pull requests.

Pull requests should meet the following criteria.

1. Fix one issue and fix it well.
2. Do not include extraneous changes that do not relate to the issue being fixed.
3. Include a descriptive title and description for the pull request.
4. Have descriptive commit messages.

Releasing ChatterBot

ChatterBot follows semantic versioning as a set of guidelines for release versions.

- **Major** releases (2.0.0, 3.0.0, etc.) are used for large, potentially backwards incompatible changes.
- **Minor** releases (2.1.0, 2.2.0, 3.1.0, 3.2.0, etc.) are used for releases that contain features and dependency changes.
- **Patch** releases (e.g., 2.1.1, 2.1.2, 3.0.1, 3.0.10, etc.) are used for releases that contain *only* bug fixes.

These rules are adhered to as much as possible but sometimes mistakes are made.

Release Process

The following procedure is used to finalize a new version of ChatterBot.

1. We make sure that all CI tests on the master branch are passing.
2. We tag the release on GitHub.
3. A new package is generated from the latest version of the master branch.

```
python setup.py sdist bdist_wheel
```

4. The Python package files are uploaded to PyPi.

```
twine upload dist/*
```

Unit Testing

"A true professional does not waste the time and money of other people by handing over software that is not reasonably free of obvious bugs; that has not undergone minimal unit testing; that does not meet the specifications and requirements; that is gold-plated with unnecessary features; or that looks like junk." – Daniel Read

ChatterBot tests

ChatterBot's built in tests can be run using nose. See the nose documentation for more information.

```
nosetests
```

Note that nose also allows you to specify individual test cases to run. For example, the following command will run all tests in the test-module *tests/logic_adapter_tests*

```
nosetests tests/logic_adapter_tests
```

Django integration tests

Tests for Django integration have been included in the *tests_django* directory and can be run with:

```
python runtests.py
```

Django example app tests

Tests for the example Django app can be run with the following command from within the *examples/django_app* directory.

```
python manage.py test
```

Benchmark tests

You can run a series of benchmark tests that test a variety of different chat bot configurations for performance by running the following command.

```
python tests/benchmarks.py
```

Makefile Utility

Makefiles are a simple way to perform code compilation on `Linux platforms`.

We often forgot to build docs, run nosetes or Django tests whenever we make any change in existing files, and when we create a pull request for the same,it fails the build giving the explanation : *Some checks were not successful*

To avoid all your problems with the Travis CI, use the `MAKEFILE`. It will help you with the code to avoid problems, failing the build by Travis CI.

To see the list of avaliable commands with MAKEFILE:

```
make help
```

To run all tests:

```
make all
```

To clean your workspace with un-versioned files

```
make clean
```

Packaging your code for ChatterBot

There are cases where developers may want to contribute code to ChatterBot but for various reasons it doesn't make sense or isn't possible to add the code to the main ChatterBot repository on GitHub.

Common reasons that code can't be contributed include:

- Licencing: It may not be possible to contribute code to ChatterBot due to a licencing restriction or a copyright.
- Demand: There needs to be a general demand from the open source community for a particular feature so that there are developers who will want to fix and improve the feature if it requires maintenance.

In addition, all code should be well documented and thoroughly tested.

Package directory structure

Suppose we want to create a new logic adapter for ChatterBot and add it the Python Package Index (PyPI) so that other developers can install it and use it. We would begin doing this by setting up a directory file the following structure.

Listing 4.1: Python Module Structure

```
IronyAdapter/
|-- README
|-- setup.py
|-- irony_adapter
|   |-- __init__.py
|   |-- logic.py
|-- tests
|-- |-- __init__.py
|-- |-- test_logic.py
```

More information on creating Python packages can be found here: https://packaging.python.org/tutorials/distributing-packages/

Register on PyPI

Create an account: https://pypi.python.org/pypi?%3Aaction=register_form

Create a `.pypirc` configuration file.

Listing 4.2: .pypirc file contents

```
[distutils]
index-servers =
pypi

[pypi]
```

```
username=my_username
password=my_password
```

Generate packages

```
python setup.py sdist bdist_wheel
```

Upload packages

The official tool for uploading Python packages is called twine. You can install twine with pip if you don't already have it installed.

```
pip install twine
```

```
twine upload dist/*
```

Install your package locally

```
cd IronyAdapter
pip install . --upgrade
```

Using your package

If you are creating a module that ChatterBot imports from a dotted module path then you can set the following in your chat bot.

```
chatbot = ChatBot(
    "My ChatBot",
    logic_adapters=[
        "irony_adapter.logic.IronyAdapter"
    ]
)
```

Testing your code

```
from unittest import TestCase


class IronyAdapterTestCase(TestCase):
    """
    Test that the irony adapter allows
    the chat bot to understand irony.
    """

    def test_irony(self):
        # TODO: Implement test logic
        self.assertTrue(response.irony)
```

Glossary

adapters A pluggable class that allows a ChatBot instance to execute some kind of functionality.

logic adapter An adapter class that allows a ChatBot instance to select a response to

storage adapter A class that allows a chat bot to store information somewhere, such as a database.

input adapter An adapter class that gets input from somewhere and provides it to the chat bot.

output adapter An adapter class that returns a chat bot's response.

corpus In linguistics, a corpus (plural corpora) or text corpus is a large and structured set of texts. They are used to do statistical analysis and hypothesis testing, checking occurrences or validating linguistic rules within a specific language territory[1].

preprocessors A member of a list of functions that can be used to modify text input that the chat bot receives before the text is passed to the logic adapter for processing.

statement A single string of text representing something that can be said.

response A single string of text that is uttered as an answer, a reply or an acknowledgement to a statement.

untrained instance An untrained instance of the chat bot has an empty database.

[1] https://en.wikipedia.org/wiki/Text_corpus

CHAPTER 5

Report an Issue

Please direct all bug reports and feature requests to the project's issue tracker on GitHub.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

S

T

U

V