Botpy Documentation

Ashish Ahuja

Nov 09, 2019

Contents

A python framework for creating bots on the StackExchange network. Builds upon ChatExchange to create a nice framework to help you make bots without the nitty-gritty you would otherwise have.

CHAPTER 1

Features

- ChatUser management, along with privilege levels. Set up privilege levels using a single line of code. Botpy also provides different privilege chat commands to get your privilege system up and running in seconds!
- Command management, which you can easily extend with your own commands. Provides a simple template to make commands. Also provides a huge number of default commands you can configure.
- Redunda support, to help you run multiple instances at the same time and also have backups of all your bot data.
- A fully functional background task manager using threads, which allows you to add your own tasks. Automatically stops and starts tasks based on the status of the instance.

CHAPTER 2

Installation

Botpy has been thoroughly tested on Python 3.6 (and *should* work on all versions above Python 3). To install the latest version from PyPi, simply run:

```
$ pip3.6 install -U BotpySE
```

or

```
$ sudo -H pip3.6 install BotpySE
```

Note: Compatibility issues with versions of python below 3.6 will not be fixed by the author.

Warning: The PyPi package is named `BotpySE`, not `Botpy`.

CHAPTER 3

License

Licensed under WTFPL (official site, license text). Use this project for whatever the fuck you want to do.

CHAPTER 4

Table of Contents

4.1 Overview

This section will provide a small introduction to headstart you understanding Botpy. At the end, we will construct a simple chatbot.

4.1.1 The Bot Object

Botpy is heavily based around one object, that is the `Bot` object.

First, you have to initialise a `Bot` which is defined as follows:

```
__init__(self, bot_name, commands, room_ids, background_tasks=[], host='stackexchange.
↪com', email=None, password=None)
```

1. `bot_name` is a string which will hold the name of the bot. This name will be used in a multitude of places, but most crucially, in recognizing commands.

1. All chat messages (in rooms the bot is in) starting with `@<bot name>` will be recognised as a command directed towards the bot.
2. If the name specified is `testbot`, then pings ranging from `@tes` to `@testbot` will be recognised. Three letter of the bot name is a minimum for recognition.
3. The case of this argument does not matter.

2. `commands` is a list consisting of `Command` objects.

1. All commands which can be run by the bot will have to be listed in this argument.
2. The command will have appropriate functions for command usage, privileges and code to be run on invocation. Look at the commands section in the User Guide for more information on writing commands.

Warning: Objects passed through this argument which do not adhere to the given command format will lead to the bot malfunctioning to an extent which cannot be predicted.

3. `room_ids` is a list consisting of room ids the bot should join.
1. The Bot will not listen to any messages and recognise any commands from rooms not specified in this list.
2. The room id must exist on the specified host (SO, SE, or MSE).
4. `background_tasks` is a list consisting of `BackgroundTask` objects.
1. The BackgroundTaskManager opens up threads for processes which will run while the bot is alive. If you have another process to run during the life of the bot, it is advised to use the BackgroundTaskManager since it will shut down and start these tasks automatically whenever required.
2. Leave this field blank (`[]`) if no extra processes are required to run during the life of the bot, or if you choose to not use the BackgroundTask Manager.
3. For more information on constructing `BackgroundTask` objects, look at the background tasks section.
5. `host` is a string consisting of the chat host the bot will run on.
   1. Possible hosts are:
   1. `stackexchange.com` (default)
   2. `stackoverflow.com`
   3. `meta.stackexchange.com`
6. `email` is a string consisting of the email of the account the bot will use (default: `None`).
7. `password` is a string consisting of the password of the account the bot will use (default: `None`).

Note: It is recommended to store your email and password in environment variables, or read them in during program execution. Please *avoid* hardcoding them in your program.

4.1.2 Running the bot

Once the bot object has been constructed, a majority of the work has been done. As you will see in further sections, using the privilege system, using Redunda, etc. will require more changes after the bot object has been created. For now though, we can simply focus on starting and stopping the bot.

```
bot.start()
```

The above line will start the bot (where `bot` is the `Bot` object) and start running all background processes specified. Processes required to keep the bot alive such as listening to rooms, running commands, etc. will automatically run.

```
bot.stop()
```

This will stop the bot. For your convenience, the bot is automatically stopped when the stop command is run. A background task runs to continuously check whether the bot has to be rebooted or stopped. Use this method to stop the bot if need be, and if you know what you're doing.

Note: If stopping the bot manually is a real necessity, it is recommended to set `Utilities.StopReason.shutdown` (code reference here) to `True` instead of directly using the `stop` function. Eventually, the `stop` function will be triggered.

4.1.3 A simple chatbot

This section consists of a simple example of a chatbot named “Testbot” using all of Botpy’s default features. The bot constructed here will have all functional commands (start, stop, reboot, alive, etc.) and will monitor and store room and user data. In this specific example, I will be using the Sandbox room on StackOverflow chat.

```
import os
import getpass
import BotpySE as bp

if "ChatbotEmail" in os.environ:
    email = os.environ["ChatbotEmail"]
else:
    email = input("Email: ")

if "ChatbotPass" in os.environ:
    password = os.environ["ChatbotPass"]
else:
    password = getpass.getpass("Password: ")

commands = bp.all_commands # We are using all of Botpy's default commands, and not␣
↪creating any of our own.

rooms = [1]                # This bot will join only one room, that is the Sandbox␣
↪room (room id 1) on StackOverflow chat.

background_tasks = []      # We will not be having any background tasks in this bot.
                           # All tasks required to keep the bot alive such as␣
↪monitoring rooms will be automatically added.

host = "stackoverflow.com" # Our chat room is on StackOverflow chat.

bot = bp.Bot("TestBot", commands, rooms, background_tasks, host, email, password)

# Erase email and password from memory.
email = ""
password = ""

# Start the bot. The bot will run forever till a stop command is run. The reboot␣
↪command will automatically reboot the bot.
# All background tasks specified and those automatically added will continue running␣
↪till the bot stops.
bot.start()
```

Before you run this bot, there is one more requirement needed to be fulfilled. Botpy stores all required user files at `~/.<bot name>`. In this case, before you run the bot, you need to create a directory. Run:

```
$ mkdir ~/.testbot
```

You’re now all good to go! Try running the bot. Go to the sandbox room on SO chat and run some commands. This is all what is required to run a fully functional chatbot on the StackExchange network.

Note: The bot account you are using needs to have at least 20 reputation on StackOverflow to chat. If you do not have a bot account, or simply do not have 20 rep in it, you can use your own account. It might be slightly weird with the bot responding to your own messages from the same account, but it'll work.

4.2 The Privilege System

Botpy has an inbuilt privilege system to allow only trusted users to run certain commands, and to keep your bot safe and secure at all times.

4.2.1 System Design

Botpy's privilege system can have an unlimited number of privilege levels. Each level is associated with a number which tells us how much power each level has. By default, there are no privilege levels, and therefore each command of the bot can be run by anyone.

For example, a bot can have two privilege levels:

Level Name	Grant Level
regular user	1
owner	2

Each user is associated with a privilege level. A user does not have a privilege level till assigned one. In other words, by default, every user's grant level is 0. All users who have a privilege level can be listed using the default `membership` command (code reference here).

Each command has a minimum grant level, which can be configured as you wish (see commands section for more information). So, if a command's minimum grant level is set to 1, but the user running the command has no privilege level (or one with the grant level lower than 1), the command will not run. For a command to be executed, the command's privilege level needs to be lesser than or equal to the user's grant level.

A user's privileges can be changed by using the

Note: If you plan on using a privilege system in your bot, it is recommended to set the grant levels of the command to privilege and unprivilege users at the highest level. Without that, users without privileges would be able to increase their grant level and run all commands, which would defeat the purpose of the privilege system.

Warning: Setting a command's grant level higher than the maximum grant level of the privilege levels will render the command unusable, even for users with the highest privileges. Do not do this.

Note: User privileges are room based. This means that a user with privileges or an RO with privileges in one room will not have privileges in another unless added.

Now, let's see how to implement the above described system through Botpy.

4.2.2 Adding a privilege level

A function named `add_privilege_type` (github reference here) which is a part of the `Bot` class is defined as follows:

```
add_privilege_type(self, privilege_level, privilege_name)
```

The first (excluding `self`) argument `privilege_level` is an integer containing the grant level. The second argument, `privilege_name` contains the name of the privilege level.

Example code to implement two privilege levels follows.

```
bot.add_privilege_type(1, "regular_user")
bot.add_privilege_type(2, "owner")
```

The two privilege types have now been added!

Note: Add privilege levels only *after* the bot has started, or else the bot will most probably crash.

4.2.3 Granting all room owners maximum privileges

Often, granting all room owners (ROs) of a room privileges makes sense. Now, instead of manually adding all ROs to the privilege list, Botpy provides a function to privilege all ROs, which is defined as follows:

```
set_room_owner_privs_max(self, ids=[])
```

The function `set_room_owner_privs_max` (reference here) will grant all ROs maximum privileges. By default, it does this for all rooms the bot runs in. If you want ROs to have privileges in specific rooms only, specify the room ids in the second argument, `ids`.

The following

```
bot.set_room_owner_privs_max()
```

will grant maximum privileges to all ROs in all rooms the bot is in!

Warning: Once this has been run, ROs in these rooms will have the privileges forever. Simply deleting this line from the bot's code will not revert this. Their privileges will have to be manually removed through bot commands.

4.3 Commands

Botpy has a large number of default commands, and has multiple provisions to include new commands created by you. You can also add upon already existing commands. We will be constructing a simple command in this section.

4.3.1 Command Design

In Botpy, each command has to be a class which builds upon the default `Command` class (class source code here). The class has three major functions which you need to configure, namely `usage`, `privileges` and `run`. Their usage will be covered in detail in the next few sections. .

All commands which the bot should listen for need to be passed in a list while constructing the `Bot` object as mentioned in the overview section. This list should contain the complete class of each command.

So far, we know that our new command should inherit from the class `Command`. Here's the code which represents this:

```
from BotpySE as bp

class CommandTest(bp.Command):
```

4.3.2 The Usage method

The `usage` method, as the name represents, defines the usage of the command, i.e what chat messages should invoke the command. This method is often called *before* the command object is initialised, so it does not have any arguments, *including* the `self` argument which every method in a class ususally does. It is also recommended to use this method as a `@staticmethod`.

A simple `usage` method defined as follows

```
@staticmethod
def usage():
    return ["test", "demo"]
```

will lead to the command being invoked whenever a user pings the bot name with message contents of either "test" or "demo".

Commands usually have multiple arguments. Some commands *require* a minimum number of requirements. Often, the command code you write may have to check whether a number of arguments have been provided or not. Now, with appropriate usage of the `usage` method, you do not need to check for these arguments; Botpy does it for you.

In the command string being returned in the `usage` method, simply specify a `*` for every argument you need. So, a method specified as follows

```
def usage():
    return ["test *", "demo *"]
```

will make Botpy invoke the command if and *only if* 1 argument is specified after the command (a chat message such as `@<name> demo abcd` will then lead to the command being invoked with `abcd` as an argument). No arguments or more than 1 argument will lead to the bot not invoking the command.

The following code

```
def usage():
    return ["test * * *", "demo * * *"]
```

will lead to the command being invoked *only if* 3 arguments are specified.

Warning: All asterisk symbols *must* be space separated from the command, from each other and from all other symbols. If the asterisks aren't space separated from each other, such as `test **`, Botpy will not recognize the asterisk as a symbol, but as a part of the command; that will lead to the command being invoked only when the chat message is :code:'@<name> test '.

Some commands can accept an unspecified number of arguments. For such cases, use `...`.

When `...` is used, it specifies than any number of arguments will lead to the command being executed, even zero.

```
def usage():
    return ["test ...", "demo ..."]
```

will lead to the command being invoked irrespective of the number of arguments provided (including zero). As before, all arguments provided will be available in the *arguments* list present in the command instance.

In cases where a minimum number of arguments are required, combine both the techniques mentioned above and use something like

```
def usage():
    return ["test * ...", "demo * ..."]
```

which will invoke the command when *at least* 1 argument is provided.

Warning: Use these methods with care; wrong usage won't lead to warnings or errors, just undefined behavior.

4.3.3 The Run Method

Every command instance must contain a method named *run*, which gets called upon command invocation.

Say, in our test command, we want to print a message to the CLI when it gets invoked. Here is what the code would look like:

```
def run(self):
    print("Hello, World!")
```

Now, when the command is invoked through chat, this message will appear in the CLI.

Warning: Not implementing the *run* method will lead to a *NotImplementedError* being raised.

Arguments provided to the command can be accessed through a list named *arguments*.

```
def run(self):
    print(self.arguments)
```

The message which invoked the command will also be present in a *Message* instance (https://github.com/Manishearth/ChatExchange/blob/master/chatexchange/messages.py#L9).

A command can have multiple aliases; in our previous example, we could invoke our command using both *test* and *demo*. To find out the index of the element in the aliases list which invoked the command, simply check *self.usage_index*.

The Botpy command manager can also be accessed: *self.command_manager*.

4.3.4 Privileged Commands

If your bot has multiple privilege levels (see the Privileges section for more information), you might want to allow some command to be run by only some users, who belong to a specific privilege level. For this, a method named *privileges* exists in each command instance.

This method returns a single integer, which corresponds to a privilege level. All users who have a privilege level with the value being *at least* this returned integer, will be allowed to execute the command.

```
def privileges(self):
    return 2
```

The above code will make the command accessable only by users who belong to a privilege level with a value of at least 2.

Note: If the *privileges* method is not created, the privilege level defaults to 0.

4.3.5 Members

```
self.command_manager = command_manager    # CommandManager Instance
self.message = message                    # chatexchange.Message Instance
self.arguments = arguments                # List consisting of provided arguments
self.usage_index = usage_index            # Index of alias invoking the command
```

4.3.6 Other Helper Functions

The Command Class consists of two helper functions.

The *post* method allows you to post a message in the chatroom in which the command was invoked.

```
def post(self, text, length_check=True)
```

text should be a string which contains the content to be posted to the chatroom. *length_check* is a boolean, which is True by default. StackExchange chat rooms have a character-limit for single line chat-messages. Set this to False if you do not want ChatExchange checking for this character-limit, or if you are posting a multi-line message.

```
def run(self):
    self.post("Hello, fellow users!")
```

Will lead to a message being posted in the chat room in which the command was invoked.

The *reply* method directly replies to the chat message which invoked the command.

```
def reply(self, text, length_check=True)
```

The usage is the same as for the *post* method.

4.3.7 An example command

```
class CommandTest(Command):
    @staticmethod
    def usage():
        return ["test ...", "demo ..."]

    def privileges(self):
        return 2

    def run(self):
        print(self.arguments)
        self.reply("Hello, fellow users!")
```