IRC API

Sha-chan

Nov 04, 2023

CONTENTS

CHAPTER

ONE

DESCRIPTION

IRC API is a small API to make Python IRC bots. This API is based on decorators to make commands like the `cog` module in `discord.py`.

You can read the code source on Github.

If you encounter any issue feel free to open one in Github.

CHAPTER

TWO

INSTALLATION

To install IRC API, you should use `pip install irc-api`.

CHAPTER

THREE

LICENCE

This project is under GNU General Public Licence v3.0 or later (GPLv3+). Please see LICENCE for more informations

CHAPTER

FOUR

CONTENT

4.1 Tutorials

4.1.1 My first IRC bot

First of all, your bot will be composed of one main file which will contain your bot itself and one or several commands files. If the targeted IRC server has a SASL auth, you'll have to give the auth parameters by the way you want.

Let's start with a minimal bot:

```
from irc_api.bot import Bot   # imports the Bot
from irc_api import commands  # imports command's decorators

my_bot = Bot(
        ('irc.exemple.com', 6697),  # host and port for IRC server
    )

@commands.command(name='hello')  # we create a new command
def cmnd_hello(bot, msg):        # the function bounded to the command
    bot.send(msg.to, f'Hello {msg.author}')

my_bot.add_command(cmnd_hello) # we add the command to the bot

my_bot.start('SuperBot')  # we start the bot; this method take the nickname in argument
```

So here we have just created a minimal bot that will connect on the channel `#general` of the `irc.exemple.com` IRC server with the nick 'SuperBot'. When someone on this channel sends `hello`, the bot will answer.

The PINGing of IRC is fully take in charge by IRC API you don't have to make a command to handle it.

Note: You can create a new class that inherits from `Bot`:

```
from irc_api.bot import Bot

class MyBot(Bot):
        my_custom_attritute = 0

my_bot = MyBot(...)

@commands.command('get')
def get_custom_attr(bot, msg):
```

(continues on next page)

(continued from previous page)

```
        bot.send(msg.to, f'{bot.my_custom_attribute}')

my_bot.add_command(get_custom_attr)
my_bot.start(...)
```

And the same goes for custom methods.

4.1.2 SASL auth

Some IRC servers required an auth. This package can handle SASL auth, you just have to give to the bot a positionnal argument:

```
from irc_api.bot import Bot # imports the Bot
from irc_api import commands # imports command's decorators

from secrets import USER, PASSWORD  # you can import the secrets by the way you want

my_bot = Bot(
        ('irc.exemple.com', 6697),  # host and port for IRC server
        channels=['#general'],      # the channels to bot will join
        auth=(USER, PASSWORD),      # the informations for SASL auth
        prefix='!'                  # the bot's prefix.
    )
```

4.1.3 Logging

You can log what the bot receive and what callbacks are triggered. You just have to import the `logging` package and to specify a log format:

```
import logging

LOG_FORMAT = "[%(levelname)s] %(message)s"
logging.basicConfig(format=LOG_FORMAT, level=logging.INFO)
```

You can change the LOG_FORMAT to suit your need.

4.1.4 Make a command

You have at your disposal a large panel of decorators to make custom commands. Let's see them all.

All your commands must take at least two parameters:

– the bot itself (an irc_api.bot.Bot instance)

– the message that triggered the bot (an irc_api.message.Message instance)

In this part, you should import the `commands` module as follow: `from irc_api import commands`.

`commands.command(name, alias)`
: This decorator allows you to make a classic command. The bot will react when the name (or one of the aliases) is at the beginning of a message.

For exemple:

```
@commands.command(name='hello', alias=('hi', 'hey'))
def greetings(bot, msg):
    bot.send(msg.to, "Hello")
```

This command will answer if the incoming message starts with 'hello', 'hi', or 'hey' (in addition of the custom prefix given to the bot).

The commands created with this decorator can also take additionnal arguments, you should precise the type of each, so the bot can convert the arguments in the right type for you. You can also use default value if your parameter is optionnal. For instance:

```
@commands.command(name='buy', alias=('shop', 'store'))
def buy(bot, msg, article_name: str, quantity: int=1):
    if not article_name in articles_list:
        bot.send(msg.to, 'Unknown article.')
    else:
        player_buy(article_name, quantity)
        bot.send(msg.to, f'{msg.author} bought {quantity} {article_name}.')
```

Let's say that the bot's prefix is '!', if the incoming message is `!buy bread` the player will buy one unity of bread idem if the incoming is `!shop bread abc` because 'abc' is not a valid `int`. You can use quotes and double-quotes to give multi-words arguments: `!store 'little piece of bread' 5` to buy 5 unity of 'little piece of bread'. And so on and so far.

commands.on(event)

This decorator allows you to go a little bit further by giving you the possibility to trigger a command on an event. An event is a function that must take only one argument: the message (an irc_api.message.Message instance), and must returns a bool instance.

You can do litteraly what you want. Let's see a little exemple:

```
@commands.on(lambda m: 'hello' in m.text.lower())
def greetings(bot, msg):  # this type of command can't take additionnal parameters
    bot.send(msg.to, f'Hello {msg.author}'.)
```

This command say 'Hello' if there is the word 'hello' in the content of a message.

You can use several `@commands.on` on one command:

```
@commands.on(lambda m: 'hello' in m.text.lower())
@commands.on(lambda m: 'superbot' in m.text.lower())
def greetings(bot, msg):
    bot.send(msg.to, f'Hello {msg.author}'.)
```

So the command is triggered only if the two given events are on True

commands.channel

This will trigger a command at each message on a specific channel. Used on it's own, it doesn't make much sense, but it can be used to complement another decorator.

Let's see an exemple with it alone:

```
@commands.channel('#bot-test')
def test(bot, msg):
    bot.send(msg.to, f'Receive: {msg.}')
```

As I said, you can combine it:

```
@commands.channel('#bot-test')
@commands.command('stat', alias=('info',))
def player_stat(bot, msg):
    bot.send(msg.to, get_stat(msg.author))  # here msg.to is equal to '#bot-test'
```

In this exemple, the command will be only available if the message has been sent in the channel `#bot-test`.

commands.user

This decorator allow to react on a specific user's name. Like `commands.channel` it can be user in addition to another decorator.

For exemple, if you want to make some admin commands, it can be useful to check who is admin before running the admin command:

```
@commands.user('AdminPseudo')
@commands.command('kick')
def user_kick(bot, msg, user_name: str):
    kick_hammer(user_name)
    bot.send(msg.to, f'{user_name} has been kicked by {msg.author}')  # here msg.
↪author is equal to 'AdminPseudo'.
```

commands.every

This decorator is different from the others. Indeed, the others allow to trigger a command on a specific event, this decorator allow to trigger a command at regular intervals. The commands define with this decorator take only one argument (instead of two): the bot.

For instance, you want your bot to send notification when some contents is posted on a website (e.g. with RSS feed) and you want to check the website each hour:

```
@commands.every(3600)  # time between calls in seconds, 3600s = 1h
def check_rss(bot):
    if is_new_content():
        bot.send('#newspaper', "There is some new contents! Check out newspaper.org␣
↪for more infos.")
```

4.1.5 Import commands into a bot

There is several ways to import commands into the bot.

Bot.add_command

This method allows you to add a single command to the bot. It takes two arguments:

– the command itself

– a bool to consider the command as documented (`True`) or not (`False`). If the command is marked as documented, it will be stored into `Bot.commands_help`

Bot.add_commands

This allows you to a list of commands. For exemple:

```
my_bot = Bot(...)
my_bot.add_commands(cmnd1, cmnd2, cmnd3, ...)
```

To marked all the given as documented, you should add the `auto_help` command to the list:

```
from irc_api.commands import auto_help

my_bot = Bot(...)
my_bot.add_commands(auto_help, cmnd1, cmnd2, cmnd3, ...)
```

Note that you can also dynamically remove commands from the bot with the `Bot.remove_command` methode. You just have to give the command name.

4.1.6 Module of commands

When you have a complex bot, it can be more readable to isolate the commands in separates modules. In each module of commands you should import the `commands` modules with: `from irc_api import commands`.

Once you've created your modules of commands, you can import them into your bot by several ways. The first one is also the easiest:

```
import cmnd1  # modules of commands
import cmnd2
import cmnd3

from irc_api.bot import Bot
from secrets import USER, PASSWORD

my_bot = Bot(
        ('irc.exemple.com', 6697),  # host and port for IRC server
        cmnd1, cmnd2, cmnd3,        # the modules of commands, you can pass as many as␣
↪you like
        channels=['#general'],      # the channels to bot will join
        auth=(USER, PASSWORD),      # the informations for SASL auth
        prefix='!'                  # the bot's prefix.
    )

my_bot.stat('SuperBot')
```

You can also decide to declare the bot and to add the command after:

```
import cmnd1  # modules of commands
import cmnd2
import cmnd3

from irc_api.bot import Bot
from secrets import USER, PASSWORD

my_bot = Bot(
        ('irc.exemple.com', 6697),  # host and port for IRC server
        channels=['#general'],      # the channels to bot will join
        auth=(USER, PASSWORD),      # the informations for SASL auth
        prefix='!'                  # the bot's prefix.
    )

my_bot.add_commands_modules(cmnd1, cmnd2, cmnd3)  # you can pass as many modules as you␣
↪like
my_bot.stat('SuperBot')
```

4.1.7 Auto-generated assistance

An auto-generated assistance is available. It allows you to have access to the command `help`. To activate the auto-generated documentation for the whole module, you just have to import `auto_help` from `commands`, you can proceed like: `from irc_api.commands import auto_help`.

To have a constructive assistance, you can add a description to your commands by passing a `desc` positionnal argument to the decorator:

```
@commands.command(name='hello', desc='Answer hello.')
def greetings(bot, msg):
    ...
```

Note that only the first decorator can have the description, the others will be ignored:

```
@commands.channel('#bot-test', desc='An ignored description.')
@commands.on(lambda m: 'hello' in m.text().lower(), desc='This description will be␣
↪stored.')
def greetings(bot, msg):
    ...
```

You can also document your function and don't fill the `desc` argument:

```
@commands.channel('#bot-test')
@commands.on(lambda m: 'hello' in m.text().lower())
def greetings(bot, msg):
    """This description will be stored. Say hello."""
    ...
```

If the both are given (docstring and `desc`), only `desc` is stored.

In the IRC chat, you can have access to the auto-generated assistance by enter: `help` (don't forget the prefix if you have set one) to have the list of all available commands or `help cmnd` where `cmnd` is the command's name. By default, only named commands are taken in charge. Feel free to make you're own assistance function. You can use `Bot.callbacks` to get all the registered commands and `Bot.commands_help` to get only the commands that are marked as documented.

4.2 Bot

Provide a Bot class that react to IRC's message and events.

4.2.1 Classes

class irc_api.bot.**Bot**(*irc_params: tuple*, **commands_modules*, *auth: tuple = ()*, *channels: list = ['#general']*, *prefix: str = ''*, *limit: int = 100*)

Watch the IRC server and handle commands.

Attributes

prefix
[str, public] The bot's prefix for named command.

irc
[irc_api.irc.IRC, public] IRC wrapper which handle communication with IRC server.

history
[irc_api.history.History, public] The messages history.

channels
[list, public] The channels the bot will listen.

auth
[tuple, public] This contains the username and the password for a SASL auth.

callbacks
[dict, public] The callbacks of the bot. This dictionnary is like {name: command} where name is the name of the command and command a BotCommand's instance.

commands_help
[dict, public] Same that `callbacks` but only with the documented commands.

threads
[list, public] A list of threads for the commands with `@api.every`.

Methods

`__init__`(*irc_params: tuple*, **commands_modules*, *auth: tuple = ()*, *channels: list = ['#general']*, *prefix: str = ''*, *limit: int = 100*)

Initialize the Bot instance.

Parameters

irc_params
[tuple] A tuple like: (host, port) to connect to the IRC server.

auth
[tuple, optionnal] Contains the IRC server informations (host, port).

channels
[list, optionnal] Contains the names of the channels on which the bot will connect.

prefix
[str] The bot's prefix for named commands.

limit
[int] The message history of the bot. By default, the bot will remind 100 messages.

*__commands_module__
[optionnal] Modules of commands that you can give to the bot at it's creation.

`add_command`(*command*, *add_to_help: bool = False*)

Add a single command to the bot.

Parameters

command
[BotCommand] The command to add to the bot.

add_to_help
[bool, optionnal] If the command should be added to the documented functions.

add_commands(**commands*)

Add a list of commands to the bot.

Parameters

***commands**
The commands' instances.

send(*target: str*, *message: str*)

Send a message to the specified target (channel or user).

Parameters

target
[str] The target of the message. It can be a channel or user (private message).

message
[str] The content of the message to send.

start(*nick: str*)

Start the bot and connect it to IRC. Handle the messages and callbacks too.

Parameters

nick
[str] The nickname of the bot.

Examples

Assuming the module was imported as follow: `from irc_api import api` You can create a bot:

```
my_bot = api.Bot(
        irc_params=(irc.exemple.com, 6697),
        channels=["#general", "#bot-test"],
        prefix="!",
        cmnd_pack1, cmnd_pack2
    )
```

class irc_api.bot.**BotCommand**(*name: str*, *func*, *events: list*, *desc: str*, *cmnd_type: int*)

Implement a bot command.

Attributes

name
[str, public] The name of the command.

func
[function, public] The function to execute when the BotCommand is called.

events
[list, public] The list of the conditions on which the BotCommand will be called.

desc
[str, public] The description of the BotCommand. By default, the function's docstring is used.

cmnd_type
[int, public] The type of the command. * if `cmnd_type = 0`, the command is triggered on an event. * if `cmnd_type = 1`, the command is a named command. * if `cmnd_type = 2`, the command is a routine automatically triggered.

bot
[irc_api.bot.Bot, public] The bot the command belongs to.

4.2.2 Functions

irc_api.bot.**parse**(*message*)

Parse the given message to detect the command and the arguments. If a command's name is 'cmnd' and the bot receive the message `cmnd arg1 arg2` this function will returns `[arg1, arg2]`. It allows to have a powerfull commands with custom arguments.

Parameters

message
[irc_api.irc.Message] The message to parse.

Returns

args_to_return
[list] The list of the given arguments in the message.

irc_api.bot.**convert**(*data*, *new_type: type*, *default=None*)

Transtype a given variable into a given type. Returns a default value in case of failure.

Parameters

data
The given data to transtype.

new_type : type

irc_api.bot.**check_args**(*func*, **input_args*)

Check if the given args fit to the function in terms of number and type.

Parameters

func
: [function] The function the user wants to run.

***input_args**
: The arguments given by the user.

Returns

converted_args
: [list] The list of the arguments with the right type. The surplus arguments are ignored.

4.3 Commands

Defines the decorators for bot commands.

4.3.1 Decorators

`@irc_api.commands.command`(*name: str*, *alias: tuple = ()*, *desc: str = ''*)
: Create a new bot's command. Note that's a decorator.

Parameters

name
: [str] The name of the command; i.e. the string by which the command will be called.

alias
: [tuple, optionnal] The others name by which the command will be called (in addition to the given name). This parameter can be left empty if the command has no alias.

desc
: [str, optionnal] This is the description of the command. It allows you to make an auto-generated documentation with this field.

Returns

decorator
: [function] This function take in argument the function you want to transform into a command and returns a BotCommand's instance.

Examples

For example, assuming the module was imported as follow: `from irc_api import commands` You can make a command:

```
@commands.command(name="ping", desc="Answer 'pong' when the user enters 'ping'.")
def foo(bot, message):
    bot.send(message.to, "pong")
```

`@irc_api.commands.`**channel**(*channel_name: str*, *desc: str = ''*)

Allow to create a command when the message come from a given channel. This decorator can be used with another one to have more complex commands.

Parameters

channel_name
[str] The channel's name on which the command will be called.

desc
[str, optionnal] This is the description of the command. It allows you to make an auto-generated documentation with this field.

Returns

decorator
[function] This function take in argument the function you want to transform into a command and returns a BotCommand's instance.

Examples

Assuming the module was imported as follow: `from irc_api import commands` If you want to react on every message on a specific channel, you can make a command like:

```
@commands.channel(channel_name="bot-test", desc="The bot will react on every␣
↪message post on #bot-test")
def spam(bot, message):
    bot.send("#bot-test", "This is MY channel.")
```

You can also cumulate this decorator with `@commands.command`, `@commands.on` and `@commands.user`:

```
@commands.channel(channel_name="bot-test") # note that the description given here␣
↪isn't taken into account
@commands.command(name="troll", desc="Some troll command")
def troll_bot(bot, message):
    emotions = ("happy", "sad", "angry")
    bot.send("#bot-test", f"*{choice(emotions)} troll's noises*")
```

`@irc_api.commands.`**every**(*time: float*, *desc=''*)

This is not a command but it allows you to call some routines at regular intervals.

Parameters

time
[float]

The time in seconds between two calls.

desc
[str, optionnal] This is the description of the command. It allows you to make an auto-generated documentation with this field.

Returns

decorator
[function] This function take in argument the function you want to transform into a command and returns a BotCommand's instance.

Examples

Assuming the module was imported as follow: `from irc_api import commands`. You can make a routine:

```
@commands.every(time=5, desc="This routine says 'hello' on #general every 5 seconds
↪")
def spam(bot, message):
    bot.send("#general", "Hello there!") # please don't do that (.><)'
```

`@irc_api.commands.on`(*event*, *desc: str = ''*)

Make a command on a custom event. It can be useful if you want to have a complex calling processus. Such as a regex recognition or a specific pattern. This decorator allows you to call a command when a specific event is verified.

You can use several `@commands.on` on one function.

Parameters

event
[function] The `event` function should take the processed message (please refer to irc_api.irc.Message for more informations) in argument and returns a bool's instance.

desc
[str, optionnal] This is the description of the command. It allows you to make an auto-generated documentation with this field.

Returns

decorator
[function] This function take in argument the function you want to transform into a command and returns a BotCommand's instance.

Examples

Assuming the module was imported as follow: `from irc_api import commands` You can make a new command:

```
@commands.on(lambda m: isinstance(re.match(r"(.*)(merci|merci␣
↪beaucoup|thx|thanks|thank you)(.*)", m.text, re.IGNORECASE), re.Match))
def thanks(bot, message):
    bot.send(message.to, f"You're welcome {message.author}! ;)")
```

`@irc_api.commands.`**user**(*user_name: str*, *desc: str = ''*)

Allow to create a command when the message come from a given user. This decorator can be used with another one to have more complex commands.

Parameters

user_name
: [str] The user's name on which the command will be called.

desc
: [str, optionnal] This is the description of the command. It allows you to make an auto-generated documentation with this field.

Returns

decorator
: [function] This function take in argument the function you want to transform into a command and returns a BotCommand's instance.

Examples

Assuming the module was imported as follow: `from irc_api import commands`. If you want to react on every message from a specific user, you can make a command like:

```
@commands.user(user_name="my_pseudo", desc="The bot will react on every message␣
↪post by my_pseudo")
def spam(bot, message):
    bot.send(message.to, "I subscribe to what my_pseudo said.")
```

You can also cumulate this decorator with `@commands.command`, `@commands.on` and `@commands.channel`:

```
@commands.user(user_name="my_pseudo")
@commands.command(name="test", desc="Some test command.")
def foo(bot, message):
    bot.send(message.to, "Test received, my_pseudo.")
```

4.4 History

Allows to save the messages history.

4.4.1 Classes

class irc_api.history.**History**(*limit: int*)

A custom queue to have access to the latest messages.

Attributes

content
[list, private] The content of the History.

limit
[int, private] The maximum number of messages that the History stored.

Methods

add(*elmnt*)

Add a new element to the History's instance. If the History is full, the oldest message is deleted.

Parameters

elmnt
The element to add.

get()

Returns the content of the History's instance.

4.5 IRC

Manage the IRC layer.

4.5.1 Classes

class irc_api.irc.**IRC**(*host: str*, *port: int*)

Manage connection to an IRC server, authentication and callbacks.

Attributes

connected
[bool, public] If the bot is connected to an IRC server or not.

socket
[ssl.SSLSocket, private] The IRC's socket.

inbox
[Queue, private] Queue of the incomming messages.

handler : Thread, private

Methods

`__init__`(*host: str*, *port: int*)
Initialize an IRC wrapper.

Parameters

host
[str] The adress of the IRC server.

port
[int] The port of the IRC server.

`connection`(*nick: str*, *auth: tuple = ()*)
Start the IRC layer. Manage authentication as well.

Parameters

nick
[str] The nickname used.

auth
[tuple, optionnal] The tuple (username, password) for a SASL authentication. Leave empty if no SASL auth is required.

`join`(*channel: str*)
Join a channel.

Parameters

channel
[str] The name of the channel to join.

`receive`()
Receive a private message.

Returns

msg
[Message] The incoming processed private message.

`send`(*raw: str*)
Wrap and encode raw message to send.

Parameters

raw
[str] The raw message to send.

`waitfor`(*condition*)
Wait for a raw message that matches the condition.

Parameters

condition
[function] `condition` is a function that must taking a raw message in parameter and returns a boolean.

Returns

msg
[str] The last message received that doesn't match the condition.

4.6 Message

Parse the raw incoming message into a Message instance.

4.6.1 Classes

`class irc_api.message.Message`(*raw: str*)
Parse the raw message in three fields: the author, the channel and the text content.

Attributes

pattern
[re.Pattern, public] The message parsing pattern.

author
[str, public] The message's author.

to
[str, public] The message's origin (channel or DM).

text
[str, public] The message's content.

PYTHON MODULE INDEX

i

INDEX

Symbols

A

B

C

E

G

H

I

J

M

O

P

R

S

U

W