

---

# **Limnoria Documentation**

**The Limnoria/Gribble/Supybot contributors**

**Jul 28, 2023**



---

## Contents

---

<b>1</b>	<b>User Guide</b>	<b>3</b>
1.1	Installing Limnoria on GNU/Linux and UNIX (FreeBSD, macOS, ...)	3
1.2	Installing Limnoria on Windows	5
1.3	Getting Started with Limnoria/Supybot	7
1.4	Configuration	12
1.5	Identifying the bot to services	15
1.6	Capabilities	19
1.7	Security in Limnoria	22
1.8	Using the HTTP server	25
1.9	Restarting the bot automatically	26
1.10	Frequently Asked Questions	28
<b>2</b>	<b>Plugin Developer Guide</b>	<b>35</b>
2.1	Writing Your First Limnoria Plugin	35
2.2	Using <code>commands.wrap</code> to parse your command's arguments	40
2.3	Style Guidelines	47
2.4	Advanced Plugin Config	49
2.5	Advanced Plugin Testing	56
2.6	Distributing plugins	61
2.7	Using Supybot's <code>utils</code> module	64
2.8	Capabilities	70
2.9	Special methods and catching events	71
2.10	Using Limnoria's HTTP server in your plugins	73
2.11	Event scheduling using <code>supybot.schedule</code>	77
2.12	Software architecture	80
2.13	Frequently Asked Questions	82
2.14	Library reference	83
<b>3</b>	<b>Contributing to Limnoria</b>	<b>119</b>
3.1	Contributing to Limnoria as a developer	119
3.2	Translating Limnoria	120
<b>4</b>	<b>Glossary</b>	<b>123</b>
<b>5</b>	<b>Indices and tables</b>	<b>125</b>
	<b>Python Module Index</b>	<b>127</b>



Limnoria is a robust (it doesn't crash), user friendly (it's easy to configure) and programmer friendly (plugins are *extremely* easy to write) Python IRC bot. It aims to be an adequate replacement for most existing IRC bots. It includes a very flexible and powerful ACL system for controlling access to commands, as well as more than 60 builtin plugins providing around 400 actual commands.

It is the successor of Supybot since 2010 and provides many new features, but keeps full compatibility with existing configurations and plugins.

Contents:



## 1.1 Installing Limnoria on GNU/Linux and UNIX (FreeBSD, macOS, ...)

This is the “easy to follow” guide to installing Limnoria.

This guide is for non-Windows operating systems. If you want to install on Windows, check out the [Windows install guide](#).

### 1.1.1 Install

#### Install using your OS’ package manager

- Debian or Ubuntu: `sudo apt-get install limnoria`

Note that stable / LTS releases may not have the latest features or bug fixes for Limnoria. If you want a newer version than what’s in the default repositories, you can enable [Backports](#) on Debian or [Unit 193’s PPA](#) on Ubuntu.

- Fedora: `sudo dnf install limnoria`
- CentOS and Red Hat Enterprise Linux: you have to first add the right EPEL repository for your CentOS/RHEL version before being able to install the package on CentOS / RHEL. Once you have, you can run the following command to install Limnoria: `sudo yum install limnoria`
- Arch Linux: You can install Limnoria from the AUR, using either [limnoria](#) (stable releases) or [limnoria-git](#) (git snapshots).
- Gentoo: `sudo emerge net-irc/limnoria`
- Guix and GuixSD: `guix package --install limnoria`

If any of the methods above works for you, skip the next section and go to [Configuration](#).

### Other operating systems (manual install)

If you followed the section above, skip this one.

### Dependencies

The only mandatory dependency is Python 3.4 or greater.

You may also install `chardet` and `feedparser`, which are used by Limnoria if they are available.

The remaining of this guide will assume you have Python 3.

### Install Python

Python will usually come by installed by default in your distribution. If not, grab the appropriate packages from the distribution's repository, or download it from <https://python.org>.

### Install Limnoria

In the next section of this guide we will use `pip`, which is a generic way of installing Python software.

### Global installation (with root access)

If you do not have root access, skip this section.

If you are logged in as root, you can remove `sudo` from the install commands.

If you want to use the testing branch which might be more up-to date BUT LESS TESTED, replace `master` with `testing` in the commands.

First, install Limnoria's optional dependencies (you can skip this step, but some features won't be available):

```
sudo python3 -m pip install -r https://raw.githubusercontent.com/ProgVal/Limnoria/  
↪master/requirements.txt --upgrade
```

Then Limnoria itself:

```
sudo python3 -m pip install limnoria --upgrade
```

If you have an error saying No module named `pip`, install `pip` using your package manager (the package is usually named `python3-pip`).

If you have an error about externally-managed-environment, you need to setup a `virtualenv` first, then re-run the commands above:

```
python3 -m venv /opt/venvs/limnoria # creates a virtualenv at the given path  
. /opt/venvs/limnoria/bin/activate # enables the virtualenv in the current shell
```

### Local installation (without root access)

If you have followed the previous section, skip this one.



Simply add `--user` to the end of both commands. First we install requirements (you can skip it, but some features won't be available) and then Limnoria itself.:

```
python3 -m pip install -r https://raw.githubusercontent.com/ProgVal/Limnoria/master/
→requirements.txt --user --upgrade
python3 -m pip install limnoria --user --upgrade
```

You might need to add `$HOME/.local/bin` to your `PATH`.:

```
echo 'PATH="$HOME/.local/bin:$PATH"' >> ~/.$(echo $SHELL|cut -d/ -f3)rc
source ~/.$(echo $SHELL|cut -d/ -f3)rc
```

If you have an error saying No module named `pip`, install `pip` using this guide: <https://pip.pypa.io/en/stable/installing/>

If you have an error about `externally-managed-environment`, you need to setup a `virtualenv` first, then re-run the commands above:

```
python3 -m venv ~/.venvs/limnoria # creates a virtualenv at the given path
. ~/.venvs/limnoria/bin/activate # enables the virtualenv in the current shell
```

## 1.1.2 Configuration

**Note:** For historical reasons, commands are called `supybot`; but they actually run Limnoria.

We are now ready to configure Limnoria. Limnoria creates quite a few auxiliary files/directories to store its runtime data. It is thus recommended to create an empty directory from which you'll be running Limnoria, to keep all the data in a nice dedicated location. For example, you may create a 'runbot' directory inside your home directory.

Now you can `cd` to your 'runbot' directory, and from within it run `supybot-wizard`, which will walk you through a series of questions to generate the bot config file.

One thing to make sure to do in the wizard, to make your life easier down the line, is to select **y** for the *Would you like to add an owner user for your bot?* question, and actually create the owner user. Remember that password, so that you can later "identify" with the bot on IRC and administer it.

Once you generate the config file, which will be named `yourbotnick.conf` (where "yourbotnick" is the nick you have chosen for your bot in the wizard), it will be placed in your 'runbot' directory. (As long as you leave the default answer to the "Where would you like to create these directories?" question.)

Now to start the bot, run, still from within the 'runbot' directory:

```
supybot yourbotnick.conf
```

And watch the magic!

For a tutorial on using and managing the bot from here on, see the [Supybook](#).

## 1.2 Installing Limnoria on Windows

This is the "easy to follow" guide to installing Limnoria. The installation documentation provided with the Limnoria distribution is really quite good already, but since people keep coming to IRC, asking a repeating pattern of questions, we thought it would be a good idea to expand it a bit to make it a little more of a "foolproof guide".

This guide is only for Windows. If you don't want to install on Windows, check out the [non-Windows install guide](#).

### 1.2.1 Install

#### Install Python

Download the latest **Python 3** installer from <https://www.python.org>, 3.11.2, as of 2023-03-14) and run it to install Python.

Installing Python is mostly clicking next, but in the next screen remember the destination directory where you installed Python. These instructions refer to it as `C:\Python311\` which is the current name on 2023-03-14. If you downloaded a newer version, replace the version number with the new one.

Then you are asked to customize your installation. Click the drive on left side of “Python” text and select “Entire feature will be installed on local hard drive”.

Now Python installs itself which may take several minutes.

Python should be now installed and you can check that the “python” command points to correct python. Open `cmd.exe` (press the Windows button on your keyboard and type “cmd.exe”) and run `where python` and the topmost entry should be `C:\Python311\python.exe`.

#### Install Limnoria

Now we are ready to install Limnoria and its requirements. Open `cmd.exe` as **Administrator** (right click it in the previous place) and run:

```
python3 -m pip install -r https://raw.githubusercontent.com/ProgVal/Limnoria/master/
requirements.txt --upgrade
python3 -m pip install limnoria --upgrade
```

We are now ready to configure Limnoria. Limnoria creates quite a few auxiliary files/directories to store its runtime data. It is thus recommended to create an empty directory from which you’ll be running Limnoria, to keep all the data in a nice dedicated location. For example, you may create a `C:\Users\<username>\runbot` for this purpose.

### 1.2.2 Configure Limnoria

---

**Note:** For historical reasons, commands are called `supybot`; but they actually run Limnoria.

---

Now you open `cmd.exe` as **normal user**, and create and `cd` into your `runbot` directory:

```
mkdir runbot
cd runbot
```

and from within it run `supybot-wizard`:

```
python3 C:\Python311\Scripts\supybot-wizard
```

which will walk you through a series of questions to generate the bot config file.

One thing to make sure to do in the wizard, to make your life easier down the line, is to select `y` for the *Would you like to add an owner user for your bot?* question, and actually create the owner user. Remember that password, so that you can later “identify” with the bot on IRC and administer it.

Once you generate the config file, which will be named `yourbotnick.conf` (where `yourbotnick` is the nick you have chosen for your bot in the wizard), it will be placed in your `runbot` directory. (As long as you leave the default answer to the *Where would you like to create these directories?* question.)

Now to start the bot, run, still from within the `C:\users\<username>\runbot` directory:

```
python3 C:\Python311\Scripts\supybot yourbotnick.conf
```

And watch the magic!

This guide has been mainly written by nanotube (Daniel Folkinshteyn), and is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported license and/or the GNU Free Documentation License v 1.3 or later.

## 1.3 Getting Started with Limnoria/Supybot

### 1.3.1 Introduction

Ok, so you've decided to try out Limnoria. That's great! The more people who use Limnoria, the more people can submit bugs and help us to make it the best IRC bot in the world :)

You should have already read through our install document (if you had to manually install) before reading any further. Now we'll give you a whirlwind tour as to how you can get Limnoria setup and use Limnoria effectively.

### 1.3.2 Initial Setup

Now that you have Limnoria installed, you'll want to get it running. The first thing you'll want to do is run `supybot-wizard`. Before running `supybot-wizard`, you should be in the directory in which you want your bot-related files to reside. The wizard will walk you through setting up a base config file for your Limnoria. Once you've completed the wizard, you will have a config file called `botname.conf`. In order to get the bot running, run `supybot botname.conf`.

### 1.3.3 Listing Commands

Ok, so let's assume your bot connected to the server and joined the channels you told it to join. For now we'll assume you named your bot 'mybot' (you probably didn't, but it'll make it much clearer in the examples that follow to assume that you did). We'll also assume that you told it to join `#channel` (a nice generic name for a channel, isn't it? :)) So what do you do with this bot that you just made to join your channel? Try this in the channel:

```
<user> supybot: list
<supybot> Admin, Channel, ChannelLogger, Config, Misc, Network, Owner, Plugin, User,
↳ and Utilities
```

Replacing 'supybot' with the actual name you picked for your bot, of course. Your bot should reply with a list of the plugins it currently has loaded. At least *Admin*, *Channel*, *Config*, *Misc*, *Owner*, and *User* should be there; if you used `supybot-wizard` to create your configuration file you may have many more plugins loaded. The list command can also be used to list the commands in a given plugin:

```
<user> supybot: list Misc
<supybot> user: apropos, clearmores, completenick, help, last, list, more, noticetell,
↳ ping, source, tell, and version
```

This listed all the commands in the *Misc* plugin. If you want to see the help for any command, just use the help command:

```
<user> supybot: help help
<supybot> user: (help [<plugin>] [<command>]) -- This command gives a useful
↳ description of what <command> does. <plugin> is only necessary if the command is in
↳ more than one plugin. You may also want to use the 'list' command to list all
↳ available plugins and commands.
<user> supybot: help list
<supybot> user: (list [--unloaded] [<plugin>]) -- Lists the commands available in the
↳ given plugin. If no plugin is given, lists the public plugins available. If --
↳ unloaded is given, it will list available plugins that are not loaded.
<user> supybot: help load
<supybot> user: (load <plugin>) -- Loads the plugin <plugin> from any of the
↳ directories in conf.supybot.directories.plugins; usually this includes the main
↳ installed directory and 'plugins' in the current directory.
```

Sometimes more than one plugin will have a given command; for instance, the “list” command exists in both the Misc and Config plugins (both loaded by default). List, in this case, defaults to the Misc plugin, but you may want to get the help for the list command in the Config plugin. In that case, you’ll want to give your command like this:

```
<user> supybot: help config list
<supybot> user: (config list <group>) -- Returns the configuration variables
↳ available under the given configuration <group>. If a variable has values under it,
↳ it is preceded by an '@' sign.
```

Anytime your bot tells you that a given command is defined in several plugins, you’ll want to use this syntax (“plugin command”) to disambiguate which plugin’s command you wish to call. For instance, if you wanted to call the Config plugin’s list command, then you’d need to say:

```
<user> supybot: config list
```

Rather than just ‘list’.

### 1.3.4 Making Limnoria Recognize You

For making the bot to identify to services, please see *identifying to services*.

If you ran the wizard, then it is almost certainly the case that you already added an owner user for yourself. If not, however, you can add one via the handy-dandy ‘supybot-adduser’ script. You’ll want to run it while the bot is not running (otherwise it could overwrite supybot-adduser’s changes to your user database before you get a chance to reload them). Just follow the prompts, and when it asks if you want to give the user any capabilities, say yes and then give yourself the ‘owner’ capability, restart the bot and you’ll be ready to load some plugins!

Now, in order for the bot to recognize you as your owner user, you’ll have to identify with the bot.

Open up a query window in your irc client (‘/query’ should do it; if not, just know that you can’t identify in a channel because it requires sending your password to the bot). Then type this:

```
<user> help identify
<supybot> (identify <name> <password>) -- Identifies the user as <name>. This command
↳ (and all other commands that include a password) must be sent to the bot privately,
↳ not in a channel.
```

And follow the instructions; the command you send will probably look like this, with ‘myowneruser’ and ‘myuser-password’ replaced:

```
<user> identify myowneruser myuserpassword
<supybot> The operation succeeded
```

The bot told you ‘The operation succeeded’, meaning that you got the right name and password. Now that you’re identified, you can do anything that requires any privilege: that includes all the commands in the Owner and Admin plugins, which you may want to take a look at (using the list and help commands, of course). One command in particular that you might want to use (it’s from the User plugin) is the ‘hostmask add’ command: it lets you add a hostmask to your user record so the bot recognizes you by your hostmask instead of requiring you always to identify with it before it recognizes you. Use the ‘help’ command to see how this command works. Here’s how I often use it:

```
<user> hostmask add myuser [hostmask] mypassword
<supybot> The operation succeeded
```

You may not have seen that ‘[hostmask]’ syntax before. Limnoria allows nested commands, which means that any command’s output can be nested as an argument to another command. The hostmask command from the User plugin returns the hostmask of a given nick, but if given no arguments, it returns the hostmask of the person giving the command. So the command above adds the hostmask I’m currently using to my user’s list of recognized hostmasks. I’m only only required to give mypassword if I’m not already identified with the bot.

It might often be better to specify the hostmask by yourself instead of nesting the hostmask command as the hostmask command gives your exact hostmask of that moment meaning `nick!ident@host` which means that you will get unidentified if you change your nickname.

I (Mikaela) often specify hostmasks in two other forms depending on the situation which I go through in next subtopics.

### Wildcard nick

In case my username and host stay the same or there aren’t bots on same server which could get identified as me to other bots, I use:

```
<user> user hostmask add myuser *!myident@myhost
<supybot> The operation succeeded
```

I only recommend this if there is ident server configured and the IRC network checks for it.

### Host only

In case I am the only one who has the same host (cloaks/vhosts on many networks which have account in them, (for example Libera) or server where no one else has access and no bots share it either), I use:

```
<user> user hostmask add myuser *!*@mycloak
<supybot> The operation succeeded
```

Mycloak at Libera is usually in format `user/accountname`. You can usually request hostmasks using HostServ, `/msg HostServ help`, or asking on help channel of your IRC network, in case of Libera that is #libera. OFTC is exception to this and uses `/msg NickServ set cloak on`, but whatever your network users, you can ask it on their help channel.

### Limnoria

Limnoria has two additional methods to identify, GPG and NickAuth, each provided as a plugin that you need to load (with the `load` command).

### GPG

First you must associate your GPG key with your Limnoria account. The `gpg add` command takes two arguments, key id and key server.

My key is 0x0C207F07B2F32B67 and it's on keyserver pool.sks-keyservers.net so and now I add it to my bot:

```
<Mikaela> +gpg add 0x0C207F07B2F32B67 pool.sks-keyservers.net
<Yvzabevn> 1 key imported, 0 unchanged, 0 not imported.
```

Now I can get token to sign so I can identify:

```
<Guest45020> +gpg gettoken
<Yvzabevn> Your token is: {03640620-97ea-4fdf-b0c3-ce8fb62f2dc5}. Please sign it with
↳ your GPG key, paste it somewhere, and call the 'auth' command with the URL to the
↳ (raw) file containing the signature.
```

Then I follow the instructions and sign my token in terminal:

```
echo "{03640620-97ea-4fdf-b0c3-ce8fb62f2dc5}" | gpg --clearsign | curl -F 'sprunge=-'
↳ http://sprunge.us
```

Note that I sent the output to curl with flags to directly send the clearsigned content to sprunge.us pastebin. Curl should be installed on most of distributions and comes with `msysgit`. If you remove the curl part, you get the output to terminal and can pastebin it to any pastebin of your choice. Sprunge.us has only plain text and is easy so I used it in this example.

And last I give the bot link to the plain text signature:

```
<Guest45020> +gpg auth http://sprunge.us/DUdd
<Yvzabevn> You are now authenticated as Mikaela.
```

### NickAuth

This requires you to load the NickAuth plugin (see next section of this page for loading plugins).

NickAuth allows you to identify to the bot using your NickServ account. First I add my NickServ account name which I can see with “/whois Mikaela Mikaela” (because my current nick is Mikaela). It gives me something like:

```
[Mikaela] is logged in as Mikaela
```

Now I tell the bot add my NickServ account Mikaela to my bot user on Libera. The syntax is [`<network>`] `<bot-username>` `<NickServ-account>`:

```
<Mikaela> +nickauth nick add Libera Mikaela Mikaela
<Yvzabevn> OK.
```

Next time when I identify to NickServ I will get identified automatically if the bot sees that I was identified when I joined. This requires server to support extended-join and WHOX. Most of modern networks support them, but if your bot is using some bouncer, it might not support them.

Automatic identification doesn't work always even when it's supported, but when it fails, I can always use the NickAuth command to identify to the bot:

```
<Guest45020> +whoami
<Yvzabevn> I don't recognize you. You can message me either of these two commands:
↳ "user identify <username> <password>" to log in or "user register <username>
↳ <password>" to register.
```

(continues on next page)

(continued from previous page)

```
<Guest45020> +nickauth auth
<Yvzabevn> You are now authenticated as Mikaela.
```

### 1.3.5 Loading Plugins

Let's take a look at loading other plugins. If you didn't use supybot-wizard, though, you might do well to try it before playing around with loading plugins yourself: each plugin has its own configure function that the wizard uses to setup the appropriate registry entries if the plugin requires any.

If you do want to play around with loading plugins, you're going to need to have the owner capability.

Remember earlier when I told you to try `help load`? That's the very command you'll be using. Basically, if you want to load, say, the Games plugin, then `load Games`. Simple, right? If you need a list of the plugins you can load, you'll have to list the directory the plugins are in (using whatever command is appropriate for your operating system, either 'ls' or 'dir').

### 1.3.6 Understanding the help syntax

The syntax of a command describes how to run a command. The syntax is given by the help command. Some examples:

**help [<plugin>] [<command>]** This is the help of command-plugin-help.

The chevrons mean you have to replace <plugin> and <command> by a plugin name and a command name.

The brackets mean the argument they wrap is **optional**.

So, the following commands are correct:

```
<user> help
<user> help PluginName
<user> help PluginName CommandName
<user> help CommandName
```

**ping takes no arguments** This is the help for command-misc-ping.

I think it is clear enough.

**join <channel> [<key>]** This is the help for command-admin-join.

It requires a channel name, and the channel key is optional.

These two commands are ok:

```
<user> join #limnoria
<user> join #limnoria MySecretKey
```

**utilities last <text> [<text> ...]** This is the help for command-utilities-last. By the way, there is another `last` command in the *Misc* plugin, which doesn't do the same thing, that's why you need to give the plugin name.

You have to give at least one argument, but you can give as many as you wish.

### 1.3.7 Getting More From Your Limnoria

Another command you might find yourself needing somewhat often is the 'more' command. The IRC protocol limits messages to 512 bytes, 60 or so of which must be devoted to some bookkeeping. Sometimes, however, Limnoria wants

to send a message that's longer than that. What it does, then, is break it into “chunks” and send the first one, following it with (X more messages) where X is how many more chunks there are. To get to these chunks, use the *more* command. One way to try is to look at the default value of *supybot.replies.genericNoCapability* – it's so long that it'll stretch across two messages:

```
<jemfinch|lambda> $config default
                    supybot.replies.genericNoCapability
<lambdaman> jemfinch|lambda: You're missing some capability
you need. This could be because you actually
possess the anti-capability for the capability
that's required of you, or because the channel
provides that anti-capability by default, or
because the global capabilities include that
anti-capability. Or, it could be because the
channel or the global defaultAllow is set to
False, meaning (1 more message)
<jemfinch|lambda> $more
<lambdaman> jemfinch|lambda: that no commands are allowed
unless explicitly in your capabilities. Either
way, you can't do what you want to do.
```

So basically, the bot keeps, for each person it sees, a list of “chunks” which are “released” one at a time by the *more* command. In fact, you can even get the more chunks for another user: if you want to see another chunk in the last command jemfinch gave, for instance, you would just say *more jemfinch* after which, his “chunks” now belong to you. So, you would just need to say *more* to continue seeing chunks from jemfinch's initial command.

### 1.3.8 Final Word

You should now have a solid foundation for using Limnoria. You can use the *list* command to see what plugins your bot has loaded and what commands are in those plugins; you can use the ‘help’ command to see how to use a specific command, and you can use the ‘more’ command to continue a long response from the bot. With these three commands, you should have a strong basis with which to discover the rest of the features of Limnoria!

Do be sure to read our other documentation and make use of the resources we provide for assistance; this website and, of course, #limnoria on irc.libera.chat if you run into any trouble!

## 1.4 Configuration

### 1.4.1 Introduction

So you've got your Limnoria up and running and there are some things you don't like about it. Fortunately for you, chances are that these things are configurable, and this document is here to tell you how to configure them.

Configuration of Limnoria is handled via the *Config* plugin, which controls runtime access to Limnoria's registry (the configuration file generated by the ‘supybot-wizard’ program you ran). The *Config* plugin provides a way to get or set variables, to list the available variables, and even to get help for certain variables. Take a moment now to read the help for each of those commands: *config*, *list*, and *help*. If you don't know how to get help on those commands, take a look at the GETTING\_STARTED document.

### 1.4.2 Configuration Registry

Now, if you're used to the Windows registry, don't worry, Limnoria's registry is completely different. For one, it's completely plain text. But there is at least one good idea in Windows' registry: hierarchical configuration.



Limnoria's configuration variables are organized in a hierarchy: variables having to do with the way Limnoria makes replies all start with *supybot.reply*; variables having to do with the way a plugin works all start with *supybot.plugins.Plugin* (where 'Plugin' is the name of the plugin in question). This hierarchy is nice because it means the user isn't inundated with hundreds of unrelated and unsorted configuration variables.

Some of the more important configuration values are located directly under the base group, *supybot*. Things like the bot's nick, its ident, etc. Along with these config values are a few subgroups that contain other values. Some of the more prominent subgroups are: *plugins* (where all the plugin-specific configuration is held), *reply* (where variables affecting the way a Limnoria makes its replies resides), *replies* (where all the specific standard replies are kept), and *directories* (where all the directories a Limnoria uses are defined). There are other subgroups as well, but these are the ones we'll use in our example.

### 1.4.3 Configuration Groups

Using the *Config* plugin, you can list values in a subgroup and get or set any of the values anywhere in the configuration hierarchy. For example, let's say you wanted to see what configuration values were under the *supybot* (the base group) hierarchy. You would simply issue this command:

```
<Mikaela> @config list supybot
<Limnoria> #alwaysJoinOnInvite, @abuse, @capabilities, @commands, @databases, @debug,
↳@directories, @drivers, @log, @networks, @nick, @plugins, @protocols, @replies,
↳@reply, @servers, defaultIgnore, defaultSocketTimeout, externalIP, flush,
↳followIdentificationThroughNickChanges, ident, language, pidFile, snarfThrottle,
↳upkeepInterval, and user
```

These are all the configuration groups and values which are under the base *supybot* group. Actually, their full names would each have a 'supybot.' prepended to them, but it is omitted in the listing in order to shorten the output. The first entries in the output are the groups (distinguished by the '@' symbol in front of them), and the rest are the configuration values. The '@' symbol (like the '#' symbol we'll discuss later) is simply a visual cue and is not actually part of the name.

### 1.4.4 Configuration Values

Okay, now that you've used the Config plugin to list configuration variables, it's time that we start looking at individual variables and their values.

The first (and perhaps most important) thing you should know about each configuration variable is that they all have an associated help string to tell you what they represent. So the first command we'll cover is `config help`. To see the help string for any value or group, simply use the `config help` command. For example, to see what this *supybot.snarfThrottle* configuration variable is all about, we'd do this:

```
<jemfinch|lambda> @config help supybot.snarfThrottle
<supybot> jemfinch|lambda: A floating point number of seconds to
    throttle snarfed URLs, in order to prevent loops between two
    bots snarfing the same URLs and having the snarfed URL in
    the output of the snarf message. (Current value: 10.0)
```

Pretty simple, eh?

Now if you're curious what the current value of a configuration variable is, you'll use the `config` command with one argument, the name of the variable you want to see the value of:

```
<jemfinch|lambda> @config supybot.reply.whenAddressedBy.chars
<supybot> jemfinch|lambda: '@'
```

To set this value, just stick an extra argument after the name:

```
<jemfinch|lambda> @config supybot.reply.whenAddressedBy.chars @$
<supybot> jemfinch|lambda: The operation succeeded.
```

Now check this out:

```
<jemfinch|lambda> $config supybot.reply.whenAddressedBy.chars
<supybot> jemfinch|lambda: '@$'
```

Note that we used ‘\$’ as our prefix character, and that the value of the configuration variable changed. If I were to use the `flush` command now, this change would be flushed to the registry file on disk (this would also happen if I made the bot quit, or pressed Ctrl-C in the terminal which the bot was running). Instead, I’ll revert the change:

```
<jemfinch|lambda> $config supybot.reply.whenAddressedBy.chars @
<supybot> jemfinch|lambda: The operation succeeded.
<jemfinch|lambda> $note that this makes no response.
```

### 1.4.5 Default Values

If you’re ever curious what the default for a given configuration variable is, use the `config default` command:

```
<jemfinch|lambda> @config default supybot.reply.whenAddressedBy.chars
<supybot> jemfinch|lambda: ''
```

Thus, to reset a configuration variable to its default value, you can simply say:

```
<jemfinch|lambda> @config setdefault supybot.reply.whenAddressedBy.chars
<supybot> jemfinch|lambda: The operation succeeded.
<jemfinch|lambda> @note that this does nothing
```

Simple, eh?

### 1.4.6 Searching the Registry

Now, let’s say you want to find all configuration variables that might be even remotely related to `opping`. For that, you’ll want the `config search` command. Check this out:

```
<Mikaela> @config search op
<Limnoria> supybot.plugins.AutoMode.op, supybot.plugins.AutoMode.halfop, supybot.
↳ plugins.ChannelStatus.topic, supybot.plugins.LinkRelay.topicSync, supybot.plugins.
↳ NoLatin1.operator, supybot.plugins.Services.ChanServ.op, supybot.plugins.Services.
↳ ChanServ.halfop, supybot.plugins.Topic, supybot.plugins.Topic.public, supybot.
↳ plugins.Topic.separator, supybot.plugins.Topic.format, (1 more message)
<Mikaela> @more
<@Limnoria> supybot.plugins.Topic.recognizeTopiclen, supybot.plugins.Topic.default,
↳ supybot.plugins.Topic.alwaysSetOnJoin, supybot.plugins.Topic.undo, supybot.plugins.
↳ Topic.undo.max, and supybot.plugins.Topic.requireManageCapability
```

Sure, it showed all the topic-related stuff in there, but it also showed you all the `op`-related stuff, too. Do note, however, that you can only see configuration variables for plugins that are currently loaded or that you loaded in the past; if you’ve never loaded a plugin there’s no way for the bot to know what configuration variables it registers.

### 1.4.7 Network- and Channel-Specific Configuration

Many configuration variables can be specific to individual channels. The *Config* plugin provides an easy way to configure something for a specific channel; for instance, in order to set the prefix chars for a specific channel, do this in that channel:

```
<jemfinch|lambda> @config channel supybot.reply.whenAddressedBy.chars !
<supybot> jemfinch|lambda: The operation succeeded.
```

That'll set the prefix chars in the channel from which the message was sent to '!'. Voila, channel-specific values! Also, note that when using the *Config* plugin's `list` command, channel-specific values are preceded by a '#' character to indicate such (similar to how '@' is used to indicate a group of values).

Similarly, many configuration variables can be specific to individual networks. This works similarly by substituting channel with network:

```
<jemfinch|lambda> @config network supybot.reply.whenAddressedBy.chars !
<supybot> jemfinch|lambda: The operation succeeded.
```

Network-specific configuration values are preceded by a ':' character. As most (if not all) channel-specific values are also network-specific, they are preceded by '#:'.

### 1.4.8 Editing the Configuration Values by Hand

**NOTE: We don't recommend this and you shouldn't ever do this, you should do everything with the commands in the Config plugin.**

Some people might like editing their registry file directly rather than manipulating all these things through the bot. For those people, we offer the `config reload` command, which reloads both registry configuration and user/channel/ignore database configuration.

Just edit the interesting files and then give the bot the `config reload` command and it'll work as expected. Do note, however, that Limnoria flushes its configuration files and database to disk every hour or so, and if this happens after you've edited your configuration files but before you reload your changes, you could lose the changes you made. To prevent this, set the `supybot.flush` value to 'Off' while editing the files, and no automatic flushing will occur.

If you cannot access the bot on IRC and your bot is running on a POSIX system, you can also send it a SIGHUP signal; it is exactly the same as `config reload` (note that the Config plugin has to be loaded to do that).

## 1.5 Identifying the bot to services

The different methods listed here are in the order how they are usually recommended by network operators.

Please also note that SASL and CertFP are only fully supported on Limnoria. Gribble has imported partial SASL support (only PLAIN).

### 1.5.1 Registering to services

You can safely jump over this section if your bot is already registered to services.

First start by checking what is the syntax for registering with `/msg nickserv help register`. It returns you something like this (Atheme 7.x):

```
NickServ: Syntax: REGISTER <password> <email-address>
```

Assuming that that is the syntax, we can register the bot with:

```
load Services
nickserv register mypassword bot@example.com
```

Or, on Limnoria versions older than 2021.06.15:

```
ircquote PRIVMSG NickServ :REGISTER mypassword bot@example.com
```

Note that the email address must be correct. Next check that `/msg nickserv info bot` doesn't say something about being unverified. If it does, go to the email address and run:

```
nickserv VERIFY nick <code from the email>
```

Now your bot should be successfully registered and you can move to setting up automatic identifying below. If you need to identify to services now, `/msg nickserv help identify` and following the syntax (I am still assuming that you are on Atheme 7.x):

```
nickserv IDENTIFY username password
```

Note: the `nickserv` command was added in Limnoria 2021.06.15. If you have an older version, you need to run something like `ircquote privmsg nickserv :register ...` instead (note the placement of the `:` after `nickserv` and before the command name).

### 1.5.2 SASL PLAIN

*To use SASL EXTERNAL, you must only configure CertFP and it's attempted automatically.* SASL PLAIN is identifying using username and password, SASL EXTERNAL is identifying by using CertFP which is explained later on this document. It doesn't need username or password to be configured.

Note that SASL isn't supported on all networks. As the only way to check if SASL is supported is either `/quote CAP LS` (which usually gets eaten by bouncers) or connecting to the network and seeing if it works, we recommend always configuring SASL and whoising the bot to see if it worked. If it didn't work, you might want to ask the network operators about their SASL support and request them to start supporting it.

SASL is widely agreed as the best method to identify to services as it identifies you before anyone (other than IRC operators) can see that you are connected. To enable SASL, simply:

```
config networks.<network>.sasl.username AccountName
config networks.<network>.sasl.password P455w0rd
```

where you of course replace `AccountName` and `P455w0rd` with your actual NickServ account name and password. Remember to replace `<network>` with the real network name like `Libera`.

### 1.5.3 CertFP

You can test if CertFP is supported by services simply by doing `/msg NickServ cert`. If you get an error about "Insufficient parameters for CERT", CertFP is supported, and if you get an error about unknown command, it's not supported.

CertFP identifies you to services using a client (SSL) certificate and naturally requires an SSL connection. It doesn't identify you as soon as SASL, but unlike SASL, it identifies you even when services return from a netsplit, unlike any other mechanism.

First you must generate a certificate, and the easiest method is probably using OpenSSL which you should have even on Windows if you installed with pip:

```
openssl req -nodes -newkey rsa:4096 -keyout <BOT>.pem -x509 -days 3650 -out <BOT>.pem
↳ -subj "/CN=<BOT>"
```

Now you should have a <BOT>.pem file in the directory where you ran the command, presumably your home directory and you only tell your bot where to find it and tell NickServ that it belongs to you. Note that you should replace <BOT> with the account name of your bot.

You have two choices, using the same certificate on all networks:

```
config protocols.irc.certfile /home/<username>/<BOT>.pem
```

or only on one or more network where it's manually configured:

```
config networks.<network>.certfile /home/<username>/<BOT>.pem
```

And lastly, you must tell the services what is your certificate fingerprint, which you can find out with:

```
openssl x509 -sha1 -noout -fingerprint -in <BOT>.pem | tr -d ':' | tr 'A-Z' 'a-z'
```

This results in something like 05dd01fedc1b821b796d0d785160f03e32f53fa8 which you tell your bot to tell services:

```
nickserv cert add 05dd01fedc1b821b796d0d785160f03e32f53fa8
```

Or if your bot identifies as you, you can do that by yourself with:

```
/msg NickServ cert add 05dd01fedc1b821b796d0d785160f03e32f53fa8
```

Remember to replace 05dd01fedc1b821b796d0d785160f03e32f53fa8 with your own fingerprint! Next time your bot connects, it should get identified automatically.

## 1.5.4 SASL ECDSA-NIST256P-CHALLENGE

First you must ECDSA key for the bot to use:

```
openssl ecparam -name prime256v1 -genkey -out <bot>_ecdsa.pem
```

and get the public key using:

```
openssl ec -noout -text -conv_form compressed -in <bot>_ecdsa.pem | grep '^pub:' -A 3
↳ | tail -n 3 | tr -d '\n:' | xxd -r -p | base64
```

After getting the public key, you must tell your bot to use it and tell services about it (just like with CertFP/SASL EXTERNAL):

```
config supybot.networks.<network>.sasl.username AccountName
config supybot.networks.<network>.sasl.ecdsa_key /home/<username>/<BOT>_ecdsa.pem
nickserv set pubkey PUBKEY_WHICH_YOU_GOT_EARLIER
```

and after reconnecting, the bot should successfully identify using SASL ECDSA-NIST256P-CHALLENGE.

*NOTE:* You can use `ecdsa pubkey` to get the public key, but you cannot generate the key pair using it as `pyecdsa` doesn't support `ecdsatool` generated keys.

### 1.5.5 Server password

Many networks support identifying using `username:password` as server password. If this is the case with your network (anything that uses a charybdis-like IRCd), this should work for you. Note that this identifies you after SASL so, your real host might be seen. To do this, simply:

```
config networks.<network>.password username:password
```

Replace `<network>` with the name of network, for example `Libera` and `username:password` with your real username and password.

## ZNC

If you wish to connect your bot to ZNC, the recommended way is:

```
config networks.<network>.ident <username>@<identifier>/<network>
config networks.<network>.password <password>
```

The identifier is free text to describe which client your Limnoria is. It came with ZNC 1.6.0 and is completely optional. `<network>` again has been there since ZNC 1.0 which is very old and has multiple security issues that have been fixed since then. You should always run the latest release.

### 1.5.6 Services plugin

The Services plugin comes with Limnoria and should be an easy way to identify your bot, but SASL is recommended over it. Start by loading Services with:

```
load Services
```

and then tell it what NickServ and ChanServ are called:

```
config network [<network>] plugins.services.nickserv NickServ
config network [<network>] plugins.services.chanserv ChanServ
```

`[<network>]` is only necessary if the message isn't sent in the network itself. Remember to replace NickServ/ChanServ with their real names if they have a different name on any network.

If you wish to ensure that your bot never contacts an user impersonating NickServ, you may specify the server name from `/MAP` command (in your IRC client), e.g. on `Libera.Chat`:

```
config network [<network>] plugins.services.nickserv NickServ@services.
config network [<network>] plugins.services.chanserv ChanServ@services.
```

Now you can set your password:

```
services password Bot P455w0rd
```

makes the bot attempt identifying as `Bot` using password `P455w0rd`. Replace them with your real nickname and password. Note that if you have multiple nicknames, you must run `services password` for them all.

If your bot happens to get a nickname that isn't configured, it won't know how to identify. You might be able to avoid this issue by loading NickCapture, (`load NickCapture`) which attempts to regain the primary nick, when it's possible, and when it regains the primary nick, the identification should work.

## 1.6 Capabilities

### 1.6.1 Introduction

Ok, some explanation of the capabilities system is probably in order. With most IRC bots (including the ones I've written myself prior to this one) "what a user can do" is set in one of two ways. On the *really* simple bots, each user has a numeric "level" and commands check to see if a user has a "high enough level" to perform some operation. On bots that are slightly more complicated, users have a list of "flags" whose meanings are hardcoded, and the bot checks to see if a user possesses the necessary flag before performing some operation. Both methods, IMO, are rather arbitrary, and force the user and the programmer to be unduly confined to less expressive constructs.

This bot is different. Every user has a set of "capabilities" that is consulted every time they give the bot a command. Commands, rather than checking for a user level of 100, or checking if the user has an 'o' flag, are instead able to check if a user has the 'owner' capability. At this point such a difference might not seem revolutionary, but at least we can already tell that this method is self-documenting, and easier for users and developers to understand what's truly going on.

### 1.6.2 User Capabilities

What the heck can these capabilities DO?

If that was all, well, the capability system would be *cool*, but not many people would say it was *awesome*. But it **is** awesome! Several things are happening behind the scenes that make it awesome, and these are things that couldn't happen if the bot was using numeric userlevels or single-character flags. First, whenever a user issues the bot a command, the command dispatcher checks to make sure the user doesn't have the "anticapability" for that command. An anticapability is a capability that, instead of saying "what a user can do", says what a user *cannot* do. It's formed rather simply by adding a dash ('-') to the beginning of a capability; 'rot13' is a capability, and '-rot13' is an anticapability.

Anyway, when a user issues the bot a command, perhaps 'calc' or 'help', the bot first checks to make sure the user doesn't have the '-calc' or the '-help' (anti)capabilities before even considering responding to the user. So commands can be turned on or off on a *per user* basis, offering fine-grained control not often (if at all!) seen in other bots. This can be further refined by limiting the (anti)capability to a command in a specific plugin or even an entire plugin. For example, the rot13 command is in the Filter plugin. If a user should be able to use another rot13 command, but not the one in the Format plugin, they would simply need to be given '-Format.rot13' anticapability. Similarly, if a user were to be banned from using the Filter plugin altogether, they would simply need to be given the '-Filter' anticapability.

### 1.6.3 Channel Capabilities

What if #linux wants completely different capabilities from #windows?

But that's not all! The capabilities system also supports *channel* capabilities, which are capabilities that only apply to a specific channel; they're of the form '#channel.capability'. Whenever a user issues a command to the bot in a channel, the command dispatcher also checks to make sure the user doesn't have the anticapability for that command *in that channel*, and if the user does, the bot won't respond to the user in the channel. Thus now, in addition to having the ability to turn individual commands on or off for an individual user, we can now turn commands on or off for an individual user on an individual channel!

So when a user 'foo' sends a command 'bar' to the bot on channel '#baz', first the bot checks to see if the user has the anticapability for the command by itself, '-bar'. If so, it errors right then and there, telling the user that they lack the 'bar' capability. If the user doesn't have that anticapability, then the bot checks to see if the user issued the command over a channel, and if so, checks to see if the user has the antichannelcapability for that command, '#baz,-bar'. If so, again, it tells the user that they lack the 'bar' capability. If neither of these anticapabilities are present, then the bot just responds to the user like normal.



### 1.6.4 Default Capabilities

So what capabilities am I dealing with already?

There are several default capabilities the bot uses. The most important of these is the ‘owner’ capability. This capability allows the person having it to use *any* command. It’s best to keep this capability reserved to people who actually have access to the shell the bot is running on. It’s so important, in fact, that the bot will not allow you to add it with a command—you’ll have to edit the users file directly to give it to someone.

There is also the ‘admin’ capability for non-owners that are highly trusted to administer the bot appropriately. They can do things such as change the bot’s nick, cause the bot to ignore a given user, make the bot join or part channels, etc. They generally cannot do administration related to channels, which is reserved for people with the next capability.

People who are to administer channels with the bot should have the ‘#channel,op’ capability—whatever channel they are to administrate, they should have that channel capability for ‘op’. For example, since I want inkedmn to be an administrator in #supybot, I’ll give them the ‘#supybot,op’ capability. This is in addition to their ‘admin’ capability, since the ‘admin’ capability doesn’t give the person having it control over channels. ‘#channel,op’ is used for such things as giving/receiving ops, kickbanning people, lobotomizing the bot, ignoring users in the channel, and managing the channel capabilities. The ‘#channel,op’ capability is also basically the equivalent of the ‘owner’ capability for capabilities involving #channel—basically anyone with the #channel,op capability is considered to have all positive capabilities and no negative capabilities for #channel.

One other globally important capability exists: ‘trusted’. This is a command that basically says “This user can be trusted not to try and crash the bot.” It allows users to call commands like ‘icalc’ in the ‘Math’ plugin, which can cause the bot to begin a calculation that could potentially never return (a calculation like ‘10\*\*10\*\*10\*\*10’). Another command that requires the ‘trusted’ capability is the ‘re’ command in the ‘Utilities’ plugin, which (due to the regular expression implementation in Python (and any other language that uses NFA regular expressions, like Perl or Ruby or Lua or ...)) which can allow a regular expression to take exponential time to process). Consider what would happen if someone gave the bot the command ‘re [format join “” s./ [dict go] /] [dict go]’ It would basically replace every character in the output of ‘dict go’ (14,896 characters!) with the entire output of ‘dict go’, resulting in 221MB of memory allocated! And that’s not even the worst example!

### 1.6.5 Managing capabilities

#### User Capabilities

User capabilities are controlled with the `admin capability <add|remove>` and `channel capability <add|remove>`. Their difference is that the first one is only restricted to those who have the admin capability.

To make user1 admin, I would run:

```
admin capability add user1 admin
```

If the bot joins on a channel where there should be ops who shouldn’t have power over any other channel, I would run:

```
channel capability add #channel user2 op
```

Note that admins cannot give anyone capability which they don’t have by themselves first, so user1 couldn’t use `channel capability add` unless they were made #channel,op first. The command:

```
admin capability add user2 #channel,op
```

has the same effect as `channel capability add`, but it requires user to have the admin capability in addition to #channel,op.

If there is abusive user who shouldn’t have op capability but still does for one reason or another, I could run either:



```
channel capability add user3 -op
```

or:

```
channel capability remove user3 op
```

Anticapabilities are checked before normal capabilities so the first command would work even if user3 still had the op capability. Removing capability which isn't given to user or channel adds anti-capability automatically.

User capabilities can be viewed with `user capabilities` command.

## Channel

Channel capabilities affect everyone on the current channel including unidentified users. They are controlled with the `channel capability <set|unset>` commands.

If I wanted to make everyone on the channel able to voice themselves or get automatically voiced by the AutoMode plugin, I would start by unsetting the default anticapability and setting the capability.:

```
channel capability unset -voice
channel capability set voice
```

Now anyone on the channel can voice themselves or if AutoMode plugin is configured to voice voiced people, they will automatically get voiced on join.

If there was unwanted plugin or plugin which output was causing spam, Games for example, I could add anticapability for it and prevent the whole plugin from being used.:

```
channel capability set -Games
```

Note that I didn't specify any separate command after Games.

## Default

Default capabilities affect everyone whether they are identified or not. They are controlled by the `owner defaultcapability <add|remove>` command and they are commonly used for preventing users from adding/removing akas, using Unix Progstats which disabling is asked about in supybot-wizard or registering to the bot using anticapabilities.:

```
defaultcapability add -aka.add
defaultcapability add -aka.remove
defaultcapability add -user.register
defaultcapability add -unix.progstats
```

To undo this I would simply do the opposite.:

```
defaultcapability remove -aka.add
defaultcapability remove -aka.remove
defaultcapability remove -user.register
defaultcapability remove -unix.progstats
```

Defaultcapabilities can be restored with either of these two commands:

```
config setdefault capabilities
config capabilities [config default capabilities]
```

### 1.6.6 Example

To make all this less abstract, here is a popular example of what capabilities are used for: disabling a plugin or command for everyone but a select group of people

Allowing only user `foo` to use the `Games` plugin, globally:

```
defaultcapability add -games
admin capability add foo games
```

And to undo it:

```
defaultcapability remove -Games
admin capability remove foo Games
```

Same, but only on `#channel`:

```
channel capability set #channel -games
channel capability add #channel foo games

channel capability unset #channel -games
channel capability remove #channel foo games
```

And to forbid only the `dice` command of the `Games` plugin instead of the entire plugin, you would use the same commands, but with `-games.dice` and `games.dice` instead of `-games` and `games`.

### 1.6.7 Final Word

From a programmer's perspective, capabilities are flexible and easy to use. Any command can check if a user has any capability, even ones not thought of when the bot was originally written. Plugins can easily add their own capabilities—it's as easy as just checking for a capability and documenting somewhere that a user needs that capability to do something.

From an user's perspective, capabilities remove a lot of the mystery and esotery of bot control, in addition to giving a bot owner absolutely finegrained control over what users are allowed to do with the bot. Additionally, defaults can be set by the bot owner for both individual channels and for the bot as a whole, letting an end-user set the policy they want the bot to follow for users that haven't yet registered in their user database. It's really a revolution!

## 1.7 Security in Limnoria

Some security features are disabled by default. We know this is arguable, but enabling them would make it quite hard to start using the bot. This guide is for people who want to enable these features to make their bot as secure as possible.

---

**Note:** Limnoria (or Gribble or Supybot) have never been audited by a security professional. We do the best we can to make it secure, but we cannot guarantee it is completely safe.

---

### 1.7.1 Trust in network operators

As you may know, by default, it is possible to do anything from IRC, including loading the Unix plugin and using the `@call` command. The only safeguard is checking the user calling the commands is authenticated as the owner of

the bot; and network operators are able to spoof hostmasks and collect your password, thus allowing them to execute commands as the owner.

Although network operators of most well-known IRC networks are not known to do that, you should be aware of that risk.

Since version 2017.10.01, there the `supybot.commands.allowShell` config variable, to prevent malicious network operators from getting shell access on your bot's computer. It defaults to `True` to make it easy for new users to install plugins using `PluginDownloader`, but it is recommended you set it to `False` if you do not care about that feature.

Finally, you can remove the `owner` user account entirely (or remove the `owner` capability for that account). This causes every privileged commands to be unavailable, so neither you nor server operators can access it. Channel-specific configuration variables can still be configured by users with the `#channel`, `op` capability (if any), but global configuration variables can only be modified by accessing the config files.

## 1.7.2 Network connections / SSL

### Background on SSL certification validation

It is often believed using SSL magically makes impossible any attack on your connection (from the bot to the server). It is true that it prevents passive eavesdropping, but other attack methods are still possible.

The main one involves man-in-the-middle, ie. someone acting as a proxy between you (your bot, in that case) and the IRC network. If certificates are not validated, the attacker can allow you to connect to itself using their own SSL certificate, and you would never know about it.

This is why it is important to check the SSL certificate of the server you connect to: an attacker cannot spoof a certificate, or the trust of a Certificate Authority in a network's certificates.

Of course, this assumes there is no bug in your SSL library, the network's, and the protocols involved.

### Certificate validation in Limnoria

Until version 2016.02.24, Limnoria did not support certificate validation. Starting from this version, it is possible, but disabled by default, in order to not break existing bots when updating.

Certificate validation can be enabled using this command:

```
@config supybot.protocols.ssl.verifyCertificates true
```

Available validation mechanisms are Certification Authorities and fingerprint checking.

### Certificate Authorities

By default, Limnoria only checks certificates using CA certificates installed on your system. However, some networks use a CA that is not trusted by your system, such as CACert.

Limnoria allows you to add a CA certificate for a network:

```
@config networks.NETWORKNAME.ssl.authorityCertificate /path/to/the/certificate.crt
```

Note that you are responsible for making sure this is the right certificate for the CA, and trust this CA to sign correctly certificates valid for the network's hostname(s).

## Fingerprint checking

Alternatively, for networks that do not use a CA, you can give Limnoria the list of fingerprints of certificates used by the network:

```
@config supybot.networks.NETWORKNAME.ssl.serverFingerprints: <fingerprint1>
↪<fingerprint2> ...
```

Adding fingerprints will disable CA verifications (useful if you do not want to trust CAs).

Note that you are responsible for giving the correct list of fingerprints.

## Supported python versions

Fingerprint checking and CA validation are available in all Python versions supported by Limnoria.

### 1.7.3 Flooding via command abuse

Limnoria answers at most one message per command, but its message can be rather long (up to about 450 to 500 characters) for even a small command.

If this is undesirable for you, you can take the following measures:

- Limit the size of a single message with `supybot.reply.mores.length`.
- Limit how many messages the `@more` command may be called to get a response to a command: `supybot.reply.mores.maximum`
- Disable large error replies with `supybot.reply.error.detailed` and `supybot.reply.error.noCapability`, and/or send them in private with `supybot.reply.error.inPrivate`.
- And check out the various variables in `supybot.abuse.flood`.

For old bot configurations, you may also want to set the `-scheduler` capability to prevent users from using the `@scheduler add` and `@scheduler repeat` commands (bot configurations created with Limnoria versions greater than 2020.05.13 already have this by default).

We also recommend you report users abusing your bot to network operators, so they take extra measures against these users if this is against their network's policy.

### 1.7.4 Hardening

By default, Limnoria exposes much of its configuration. This is by design, to improve discoverability and debugging.

Again, if this is undesirable to you, you can do the following:

- Prevent users from using the Config plugin to read the configuration: `defaultcapability add -config` (note that sensitive configuration variables are, of course, always hidden from users by default).
- Prevent users from listing available plugins and commands: `defaultcapability add -misc.list`, `defaultcapability add -misc.apropos`, `defaultcapability add -plugin`, and `defaultcapability add -status.commands`
- Hide the version from users: `defaultcapability add -misc.version`, and also make sure it's not in `supybot.user`, `supybot.plugins.Owner.quitMsg` or `supybot.plugins.Channel.partMsg`.
- Hide capabilities users are missing to run a command: `supybot.reply.error.noCapability`

- Replace errors with a generic reply: `supybot.reply.error.detailed`

Note that, when asking for help involving an error, you should enable verbose errors when providing logs (ie. reset these last values to their default), so it is easier to help you diagnose your problems.

Finally, if you use *the systemd unit*, you can add this to its `[Service]` section:

```
SystemCallFilter=~@raw-io @clock @cpu-emulation @debug @keyring @module @mount_
↪@obsolete @privileged @raw-io
ProtectSystem=strict
ProtectHome=read-only
ReadWritePaths=/home/bot/botname
```

This might break some plugins, but most will work. You will get explicit errors if this is an issue, and you can always revert back.

## 1.8 Using the HTTP server

### 1.8.1 Configuration

The HTTP server comes with a couple of additional variables:

- `supybot.servers.http.favicon`: Path to the file which is shown to browsers as favicon.
- `supybot.servers.http.hosts4`: The IPv4 addresses where the bot will bind. In most of the cases, you will use 0.0.0.0 (everything) or 127.0.0.1 (restricted to local connections). Defaults to 0.0.0.0
- `supybot.servers.http.hosts6`: The IPv6 addresses where the bot will bind. Defaults to empty.
- `supybot.servers.http.keepAlive`: Determines whether the HTTP server will run even if has nothing to serve. Defaults to False, because the daemon might require changing the port, if it is already taken.
- `supybot.servers.http.port`: The port the bot will bind. May not work if the number is below 1024. Defaults to 8080 (alternative HTTP port).

### 1.8.2 Using the server

At the root of the server, you will find a list of the plugins that have a Web interface, and a link to them. Each plugin has one or more subdirectories of its own.

You may also want to run Apache httpd or Nginx in front of Limnoria's HTTP server, if you want to use subdomains or load balancing.

Here is an example of Apache httpd configuration (I didn't test it with the rewrite, please notify me whether it works or not):

```
<VirtualHost 0.0.0.0:80>
    ServerName stats.yourdomain.org
    <Location />
        ProxyPass http://localhost:8080/webstats/
        SetEnv force-proxy-request-1.0 1
        SetEnv proxy-nokeepalive 1
    </Location>
</VirtualHost>
```

Here is an example of the Nginx configuration. Create a new site `/etc/nginx/sites-enabled/bot`:

```
server {
    # Note that your default server should specify these ports
    listen 80;
    listen [::]:80;
    # If your default server also has HTTPS configured, uncomment
    # the following two listen lines to enable it for this vhost.
    #listen 443;
    #listen [::]:443;
    server_name stats.yourdomain.org;

    location / {
        proxy_pass http://localhost:8080/;
    }
}
```

Note that any HTTP server which can provide a reverse proxy service can be configured to act as an intermediary or front end for the Limnoria HTTP server. Configuring these alternatives is left as an exercise to the system administrator (who ought to be familiar enough with it already).

### 1.8.3 Templates

Among the plugins which use the HTTP server, some use the standard templates system which allows you to edit page templates in a standard way (for other plugins, check their documentation).

Templates are located in the *data/web/* folder. There is a folder per plugin (and a *generic* folder, which holds generic pages), and all file names end with *.example*, which is the default template provided by the plugin. To customize it, rename it to remove *.example* (for instance: `mv fooplugin/foopage.html.example fooplugin/foopage.html`) and edit it (either do it intuitively or check the plugin documentation to see how it handles its templates).

## 1.9 Restarting the bot automatically

This page documents the different ways to automatically restart your bot in case of crash or system reboot or anything that can make the bot quit.

We recommend the systemd service, if possible. If you do not have systemd as init daemon (typically when on Windows, macOS, or BSDs), skip to *supybot-botchk*.

### 1.9.1 systemd service

Using a systemd service is the recommended method to run Limnoria. You need root access as no one has got this to work as user service yet. You must also use systemd as your init system (this is usually the case on Linux).

Create a new file `/etc/systemd/system/<BOTNAME>.service` with the following content replacing things were suitable:

```
[Unit]
Description=Limnoria
After=network.target

[Service]
Environment="PATH=/usr/local/bin:/usr/local/sbin:/usr/local/games:/usr/bin:/usr/sbin:/
↪usr/games:/bin:/sbin:/bin:/opt/local/bin:/opt/local/sbin:/opt/local/games"
```

(continues on next page)

(continued from previous page)

```
Environment="TZ=UTC"
Type=simple
ExecStart=/usr/bin/supybot /home/bot/botname/botname.conf
ExecReload=/bin/kill -HUP $MAINPID
Restart=always
User=BOTUSERNAME
SyslogIdentifier=Limnoria

[Install]
WantedBy=multi-user.target
```

`/usr/bin/supybot` should be the path where you installed Limnoria. Typically, this is:

- `/usr/bin/supybot` if installed with a system package manager (APT, YUM, ...)
- `/usr/local/bin/supybot` if installed as root without a virtualenv
- `/opt/venvs/limnoria/bin/supybot` if installed as root with a virtualenv as `/opt/venvs/limnoria/`
- `/home/BOTUSERNAME/.local/bin/supybot` if installed as non-root without a virtualenv
- `/home/BOTUSERNAME/.venvs/limnoria/bin/supybot` if installed as non-root with a virtualenv as `~/.venvs/limnoria/`

Now you should run `systemctl daemon-reload` to make systemd aware of changed files and `systemctl enable <BOTNAME>.service` to make the bot start on boot etc. and `systemctl start <BOTNAME>.service` to start the bot.

Remember to check the Environment line. You can get your PATH with `printf 'PATH=%s\n' "$PATH"`.

## Some commands

- autostart on boot: `systemctl enable <BOTNAME>.service`
- disable autostart on boot: `systemctl disable <BOTNAME>.service`
- start the bot: `systemctl start <BOTNAME>.service`
- stop the bot: `systemctl stop <BOTNAME>.service`
- reload config files: `systemctl reload <BOTNAME>.service`
- show the latest logs: `journalctl -fu <BOTNAME>.service`

## 1.9.2 supybot-botchk

`supybot-botchk` is a script that comes with Limnoria which restarts the bot if it quits or system reboots or anything that causes the bot to quit. It's placed to crontab so cron will run it with scheduled intervals.

## How to use it?

## Configuring the bot

Start by telling your bot to write a pidfile somewhere where it can write, and restart the bot. For example:

```
config supybot.pidfile /home/<username>/<bot>/<bot>.pid
```

where <username> is replaced with the system username and <bot> is replaced with the name of the bot.

### crontab

After the pidfile is configured, you can modify crontab. First you should copy the output of:

```
printf 'PATH=%s\n' "$PATH"
```

and open crontab with `EDITOR=nano crontab -e` and paste the output of previous command to the first lines which don't have comments. This should be on top. You will probably also want to configure locale and timezone which happens by adding the following lines:

```
# Replace en_US.utf8 with your own locale! You should see list of
# available locales with `locale` command, just use something which
# ends with "utf8" or "UTF-8" (the latter is required on some operating
# systems like OS X).
LC_ALL=en_US.UTF-8

# Specifying timezone is optional, but you probably want to do it if
# your system is on different timezone. Replace ``UTC`` with
# ``Area/Region`` as it appears in IANA Time Zone Database if you don't
# want to use UTC.
TZ=UTC
```

NOTE: Lines starting with # are comments and don't need to be written.

Now you finally add the bot. If you have multiple bots, simply add separate lines for them all:

```
*/5 * * * * supybot-botchk --botdir=/home/<username>/<bot>/ --pidfile=/home/<username>
→ /<bot>/<bot>.pid --conffile=/home/<username>/<bot>/<bot>.conf
```

If you needed to use different environment for other bot, you could specify that on the same line. For example, my other bot uses `en_US.utf8` as locale and `UTC` as timezone:

```
*/5 * * * * LC_ALL=en_US.UTF-8 TZ=UTC supybot-botchk --botdir=/home/<username>/<bot2>/
→ --pidfile=/home/<username>/<bot2>/<bot2>.pid --conffile=/home/<username>/<bot2>/
→ <bot2>.conf
```

Note that environment doesn't need to be specified on `supybot-botchk` line unless it differs from globally specified environment which we added as the first thing to crontab.

Now you can save the crontab by pressing `CTRL + O` answering `y` and then quitting nano with `CTRL + X`.

If you are wondering what `*/5 * * * *` means, it simply means “run this every five minutes every day”. The 5 can be replaced with any other number and there are also `@hourly` etc. which can be used on its place, but you most likely won't want to wait hour or more if your bot crashes.

## 1.10 Frequently Asked Questions

This section tries to cover all questions you may have as a Limnoria user or administrator. (For questions about plugin development, check out the [Developer FAQ](#) instead.)



### 1.10.1 How do I make my Limnoria connect to multiple servers?

Just use the *connect* command in the *Network* plugin.

### 1.10.2 Why does my bot not recognize me or tell me that I don't have the 'owner' capability?

Because you've not given it anything to recognize you from!

You'll need to identify to the bot (`help identify` to see how that works) or add your hostmask to your user record (`help hostmask add` to see how that works) for it to know that you're you.

You may wish to note that `hostmask add` can accept a password; rather than identify, you can send the command:

```
hostmask add myOwnerUser [hostmask] myOwnerUserPassword
```

and the bot will add your current hostmask to your owner user (of course, you should change myOwnerUser and myOwnerUserPassword appropriately for your bot).

For additional ways to identify to your bot, you may want to see [Getting Started with Limnoria/Supybot](#).

### 1.10.3 What is a hostmask?

Each user on IRC is uniquely identified by a string which we call a *hostmask*. The IRC specifications refers to it as a "prefix" or "source". Either way, it consists of a nick, a user, and a host, in the form `nick!user@host`. If your Limnoria complains that something you've given to it isn't a hostmask, make sure that you have those three components and that they're joined in the appropriate manner.

### 1.10.4 My bot can't handle nicks with brackets in them!

It always complains about something not being a valid command, or about spurious or missing right brackets, etc.

You should quote arguments (using double quotes, like this: `"foo[bar]"`) that have brackets in them that you don't wish to be evaluated as nested commands. Alternatively, you can turn off nested commands by setting `supybot.commands.nested` to `False`, or change the brackets that nest commands by setting `supybot.commands.nested.brackets` to some other value (like `<>`, which can't occur in IRC nicks).

### 1.10.5 How do I create a command?

You can create simple commands with the Aka plugin, like this:

```
<admin> @aka add "rules" "echo Here are the rules of the channel."
<bot> Ok.
[...]
<user> @rules
<bot> Here are the rules of the channel.
```

You can also make the bot reply on arbitrary words, `MessageParser`:

```
<admin> @messageparser add "some words" "echo Blah blah"
<bot> Ok
[...]
<user> I am saying some words.
<bot> Blah blah
```

Both these examples assume you have the Utilities plugin loaded (it provides the `echo` command).

See the help of `aka add`, `messageparser add`, and `echo` to see more advanced uses of these commands (command arguments, regular expressions, variables, etc.)

While powerful, Aka and MessageParser cannot do everything. For the most advanced commands, you will need to *write your own plugin in Python*.

### 1.10.6 I loaded Alias before, how do I move to Aka?

First load both of the plugins, Aka and Alias. Then run `aka importaliasdatabase` and unload Alias. Now all your aliases should be imported to the Aka plugin.

### 1.10.7 I added an aka, but it doesn't work!

Take a look at `aka show <aka you added>`. If the aka the bot has listed doesn't match what you're giving it, chances are you need to quote your aka in order for the brackets not to be evaluated. For instance, if you're adding an aka to give you a link to your homepage, you need to say:

```
aka add mylink "format concat https://example.com/ [urlquote $1]"
```

and not:

```
aka add mylink format concat https://example.com/ [urlquote $1]
```

The first version works; the second version will always return the same url.

### 1.10.8 What does 'lobotomized' mean?

I see this word in commands and in my *channels.conf*, but I don't know what it means. What does Limnoria mean when it says lobotomized?

A lobotomy is an operation that removes the frontal lobe of the brain, the part that does most of a person's thinking. To lobotomize a bot is to tell it to stop thinking—thus, a lobotomized bot will not respond to anything said by anyone other than its owner in whichever channels it is lobotomized.

The term is certainly suboptimal, but remains in use because it was historically used by certain other IRC bots, and we wanted to ease the transition to Limnoria from those bots by reusing as much terminology as possible.

### 1.10.9 Is there a way to load all the plugins Limnoria has?

No, there isn't. Even if there were, some plugins conflict with other plugins, so it wouldn't make much sense to load them. For instance, what would a bot do with *Factoids*, *MoobotFactoids*, and *Infobot* all loaded? Probably just annoy people :)

You can also install user-contributed plugins using the `PluginDownloader` plugin (`load PluginDownloader`). The `repolist` command can list repositories and their contents, and the `install` command installs plugins.

#### 1.10.10 Is there a command that can tell me what capability another command requires?

No, there isn't, and there probably never will be.

Commands have the flexibility to check any capabilities they wish to check; while this flexibility is useful, it also makes it hard to guess what capability a certain command requires. We could make a solution that would work in a large majority of cases, but it wouldn't (and couldn't!) be absolutely correct in all circumstances, and since we're anal and we hate doing things halfway, we probably won't ever add this partial solution.

#### 1.10.11 Why doesn't *Karma* seem to work for me?

*Karma*, by default, doesn't acknowledge karma updates. If you check the karma of whatever you increased/decreased, you'll note that your increment or decrement still took place. If you'd rather *Karma* acknowledge karma updates, change the `supybot.plugins.Karma.response` configuration variable to "True".

#### 1.10.12 Why won't Limnoria respond to private messages?

The most likely cause is that your bot has a mode blocking messages from unregistered users. Around Sept. 2005, for example, Freenode added a user mode where registered users could set +R, which blocks private messages from unregistered users. So, the reason you aren't seeing a response from your Limnoria is likely:

- Your Limnoria is not registered with NickServ, you are registered, and you have set the +R user mode for yourself.
- or: you have registered your Limnoria with NickServ, you aren't registered, and your Limnoria has the +R user mode set.

#### 1.10.13 Can users with the admin capability change the configuration?

Currently, no. Feel free to make your case to us as to why a certain configuration variable should only require the *admin* capability instead of the *owner* capability, and if we agree with you, we'll change it for the next release.

#### 1.10.14 How can I make my Limnoria log my IRC channel?

To log all the channels your Limnoria is in, simply load the *ChannelLogger* plugin, which is included in the main distribution.

#### 1.10.15 Can Limnoria connect through a proxy server?

Limnoria can connect to specific network using socks proxy, simply set the configuration variable `supybot.networks.<network>.socksproxy`. For specifying proxy which is used for HTTP requests, set the configuration variable `supybot.protocols.http.proxy`.

Limnoria also works with transparent proxy server helpers like [tsocks](#) that are designed to proxy-enable all network applications, and Limnoria does work with these.

### 1.10.16 Why can't Limnoria find the plugin I want to load?

Why does my bot say that 'No plugin "foo" exists.' when I try to load the foo plugin?

First, make sure you are typing the plugin name correctly. `@load foo` may not be the same as `@load Foo` depending on your Limnoria version<sup>1</sup>. If that is not the problem,

### 1.10.17 I've found a bug, what do I do?

Submit your bug at our [issue tracker](#).

### 1.10.18 Is Python installed?

I run Windows, and I'm not sure if Python is installed on my computer. How can I find out for sure?

Python isn't commonly installed by default on Windows computers. If you don't see it in your start menu somewhere, it's probably not installed.

The easiest way to find out if Python is installed is simply to [download it](#) and try to install it. If the installer complains, you probably already have it installed. If it doesn't, well, now you have Python installed.

### 1.10.19 How can I make the bot announce titles of URLs (links) posted in channels

This is called the "title snarfer". You can enable it with:

```
load Web
config supybot.plugins.Web.titleSnarfer True
```

If you only want it for some channels but not all, use this instead of the last command:

```
config channel #channel supybot.plugins.Web.titleSnarfer True
```

### 1.10.20 Why doesn't the title snarfer announce links from a particular website (eg. Youtube)?

Limnoria needs to fetch pages to get their title. But in order to avoid being overloaded by users, it only fetches the beginning (the first 8kB if I recall correctly). That's enough to find the title of most pages, but in the last years Youtube has become so bloated it isn't.

If you are ok with Limnoria fetching more data when users post URLs, you can use:

```
config supybot.protocols.http.peekSize 300000
```

This will make it fetch 300kB from every link, instead of the default 8kB. This should be enough for Youtube for now. If not enough for other websites, try increasing it further.

---

<sup>1</sup> Yes, it used to be the same, but then we moved to using directories for plugins instead of a single file. Apparently, that makes a difference to Python.

### 1.10.21 Can I make Limnoria silent, but still working on channel (as titlesnarfer or something)?

With lobotomy, the bot stops doing everything on the channel. If you want it to not reply to commands, but still work as titlesnarfer or similar, you can configure it to not respond to anything.

Globally:

```
config supybot.reply.whenAddressedBy.chars ""
config supybot.reply.whenAddressedBy.nicks ""
config supybot.reply.whenAddressedBy.strings ""
config supybot.reply.whenAddressedBy.nick False
config supybot.reply.whenAddressedBy.nick.atEnd False
```

Or just for one channel:

```
config channel #channel supybot.reply.whenAddressedBy.chars ""
config channel #channel supybot.reply.whenAddressedBy.nicks ""
config channel #channel supybot.reply.whenAddressedBy.strings ""
config channel #channel supybot.reply.whenAddressedBy.nick False
config channel #channel supybot.reply.whenAddressedBy.nick.atEnd False
```

### 1.10.22 How to make a connection secure?

First, you should make the bot use SSL for each network:

```
config supybot.networks.<NETWORK>.ssl on
```

Then, you must update the server port for the network the bot connects to (this is usually 6697, but some networks use a different one):

```
config supybot.networks.<NETWORK>.servers irc.network.com:6697
```

In the previous command, you must of course replace *irc.network.com* with the hostname of a server of the network. You could either check out the network's website, or get the current one, with this command:

```
config supybot.networks.<NETWORK>.servers
```



## 2.1 Writing Your First Limnoria Plugin

### 2.1.1 Introduction

This page is a top-down guide on how to write new plugins for Limnoria.

Before you start, you should be more-or-less familiar with how to use and manage a Limnoria instance (loading plugins, configuring options, etc.). You should also install a copy of Limnoria on the machine you intend to develop plugins on, as it includes some additional scripts like **supybot-plugin-create** to generate the plugin skeleton.

We'll go through this tutorial by actually writing a new plugin, named Random with just a few commands.

### 2.1.2 Generating the Plugin template

The recommended way to start writing a plugin is to use the **supybot-plugin-create** wizard. You can run this from within your bot's plugins directory, or make a separate directory for all your own plugins and run it there. (You can add additional plugin directories to your bot config using `config directories.plugins`). The latter approach is probably easier if you intend to publish your code afterwards, as it keeps your code separate from any other plugins you've installed.

Here's an example session:

```
$ supybot-plugin-create
What should the name of the plugin be? Random

Sometimes you'll want a callback to be threaded. If its methods
(command or regexp-based, either one) will take a significant amount
of time to run, you'll want to thread them so they don't block the
entire bot.

Does your plugin need to be threaded? [y/n] n
```

(continues on next page)

(continued from previous page)

```
What is your name, so I can fill in the copyright and license
appropriately? John Doe
```

```
Do you wish to use Supybot's license for your plugin? [y/n] y
```

```
Please provide a short description of the plugin: This plugin contains
commands relating to random numbers, including random sampling from a list
and a simple dice roller.
```

### 2.1.3 README.md

This is the README page people will see when they download your plugin or view it from a source control website. It's helpful to include a brief summary of what the plugin does here, as well as list any third-party dependencies.

The **supybot-plugin-create** wizard should have already filled in the README with the summary you provided.

### 2.1.4 \_\_init\_\_.py

The next file we'll look at is `__init__.py`. If you're not so familiar with the Python import mechanism, think of it as sort of the "glue" file that pulls all the files in the plugin directory together when you load it. There are also a few administrative items here that can be queried from the bot, such as the plugin's author and contact info.

At the top of the file you'll see the copyright header, with your name added as prompted in **supybot-plugin-create**. Feel free to use whatever license you choose: the default is the bot's 3-clause BSD. For our example, we'll leave it as is.

Here is a list of attributes you should usually look at:

- `__version__`: the plugin version. We'll make ours "0.1"
- `__author__` should be an instance of the `supybot.Author` class. This optionally includes a full name, a short name (usually IRC nick), and an e-mail address:

```
__author__ = supybot.Author(name='Daniel DiPaolo', nick='Strike',
                             email='somewhere@someplace.xxx')
```

- `__contributors__` is a dictionary mapping `supybot.Author` instances to lists of things they contributed. See e.g. [in the Plugin plugin](#). For now we have no contributors, so we'll leave it blank.
- `__url__` references the download URL for the plugin. Since this is just an example, we'll leave this blank.

The rest of `__init__.py` shouldn't be touched unless you are using third-party modules in your plugin. If you are, then you need to add additional import statements and `reload` calls to all those modules, so that they get reloaded with the rest of the plugin:

```
from . import config
from . import plugin
from importlib import reload
reload(plugin) # In case we're being reloaded.
# Add more reloads here if you add third-party modules and want them
# to be reloaded when this plugin is reloaded. Don't forget to
# import them as well!
```



## 2.1.5 config.py

`config.py` is, unsurprisingly, where all the configuration stuff related to your plugin goes. For this tutorial, the Random plugin is simple enough that it doesn't need any config variables, so this file can be left as is.

To briefly outline this file's structure: the `configure` function is used by the **supybot-wizard** wizard and allows users to configure the plugin further if it's present when the bot is first installed. (In practice though, this is seldomly used by third-party plugins as they're generally installed *after* configuring the bot.)

The following line registers an entry for the plugin in Limnoria's config registry, followed by any configuration groups and variable definitions:

```
Random = conf.registerPlugin('Random')
# This is where your configuration variables (if any) should go. For example:
# conf.registerGlobalValue(Random, 'someConfigVariableName',
#     registry.Boolean(False, _("""Help for someConfigVariableName.""")))
```

Writing plugin configuration is explained in depth in the *Advanced Plugin Config Tutorial*.

## 2.1.6 plugin.py

`plugin.py` includes the core code for the plugin. For most plugins this will include command handlers, as well as anything else that's relevant to its particular use case (database queries, *HTTP server endpoints*, *IRC command triggers*, etc.)

As with any Python module, you'll need to import any dependencies you want, in addition to the standard `supybot` imports included in the plugin template:

```
import random
```

The bulk of the plugin definition then resides in a subclass of `callbacks.Plugin`. By convention, the class name is equal to the name of the plugin, though this is not strictly required (the actual linkage is done by the `Class = Random` statement at the end of the file). It is helpful to fill in the plugin docstring with some more details on how to actually use the plugin too: this info can be shown on a live bot using the plugin help `<plugin name>` command.

```
class Random(callbacks.Plugin):
    """This plugin contains commands relating to random numbers, including random_
    ↪ sampling from a list and a simple dice roller."""

    def __init__(self, irc):
        # Make sure to call the superclass' constructor when you define a custom one
        super().__init__(irc)
        self.rng = random.Random() # create our rng
        self.rng.seed() # automatically seeds with current time
```

For this sample plugin, we define a custom constructor (`__init__`) that instantiates a random number generator instance and pre-seeds it. This isn't technically necessary for Python's `random` module, but it helps outline how to write a similar constructor. Notice in particular how you must pass in an `irc` argument in addition to `self`.

**Warning:** Because Limnoria is a multi-network bot, you should generally ignore the `irc` instance passed to the plugin constructor. On a manual load call to a live bot, this will be set to the network the command was run on, but on bot startup, `irc` will be (arbitrarily) set to the first network that the bot decides to connect to.

### Basic command handler

Our first command definition can immediately follow:

---

**Note:** All functions used as commands must have an all lowercase name.

---

A command function taking in no arguments from IRC will still require 4 arguments; they are as follows:

- `self`: refers to the class instance. It is common to keep local state for the plugin as instance variables within the plugin class.
- `irc`: a `supybot.callbacks.ReplyIrcProxy` instance; refers to the IRC network instance the command was called on
- `msg`: a `supybot.ircmsgs.IrcMsg` instance; refers to the IRC message that triggered this command.
- `args`: a raw list of remaining unconverted arguments; new plugins that use `@wrap` for automatic argument type conversion should never need to interact with `args` directly.

The function docstring is expected to be in a particular format. First, the very first line dictates the argument list to be displayed when someone calls the `help` command on this command (i.e., `help random`). Then, leave a blank line and start the actual help string for the function. Don't worry about the fact that it's tabbed in or anything like that, as the `help` command normalizes it to make it look nice. This part should be fairly brief but sufficient to explain the function and what (if any) arguments it requires. Remember that this should fit in one IRC message which is typically around a 450 character limit.

The `irc.reply` call is a bit of magic: it issues a reply the same place as the message that triggered the command. i.e. this may be in a channel or in a private conversation with the bot.

Lastly, notice that commands go through the `@wrap` decorator for automatic argument type conversion. For commands that require no parameters, calling `@wrap` with no arguments is enough.

### Command handler with parameters

Now let's create a command with some arguments and see how we use those in our plugin commands. This `seed` command lets the user pick a specific RNG seed:

For functions that use `@wrap` (described further in the [Using commands.wrap tutorial](#)), additional command arguments are handled by:

1. Adding *type converters*, one for each parameter, to the list passed into `@wrap`
2. Adding one function parameter per argument to the command function definition. (i.e. `def seed(self, irc, msg, args, seed)` instead of `def seed(self, irc, msg, args)`)

We also modify the docstring to document this function. Note the syntax on the first line: by convention, required arguments go in `<>` and optional arguments should be surrounded by `[]`.

The function body includes a new method `irc.replySuccess`. This is just a generic "I succeeded" command which responds with whatever the bot owner has configured in `config supybot.replies.success`. Also, by using `@wrap`, we don't need to do any type checking inside the function itself - this is handled separately, and invalid argument values will cause the command to error before it reaches the wrapped function.

With this alone you'd be able to make a range of useful plugin commands, but we'll go include some more examples to illustrate common patterns.

## Command handler with list-type arguments

The next sample command is named `sample` (no pun intended): it takes a random sample of arbitrary size from a list provided by the user:

The important thing to note is that list type arguments are rolled into one parameter in the command function by the `many` filter. Similar “multiplicity” handlers are documented [here](#).

We also update the docstring to use the `[]` convention when surrounding optional arguments.

For this function’s body, `irc.error` is like `irc.replySuccess` but for error messages. We prefer using this instead of `irc.reply` for error signaling because its behaviour can be configured specially. For example, you can force all errors to go in private by setting the `reply.error.inPrivate` option, and this can help reduce noise on a busy channel. Also, `irc.error()` with no text will return a generic error message configured in `supybot.replies.error`, but this is not a valid call to `irc.reply`.

`utils.str.commaAndify` is a simple helper that takes a list of strings and turns it into “item1, item2, item3, item4, and item5” for an arbitrary length. Limnoria has accumulated many such helpers in its lifetime, many of which are described in the [Using Utils](#) page.

## Command handler with optional arguments

Now for the last command that we will add to our `plugin.py`. This `diceroll` command will allow the bot users to roll an arbitrary n-sided die, with n defaulting to 6:

The only new thing described here is that `irc.reply(..., action=True)` makes the bot perform a */me*. There are some other flags described in the [irc.reply](#) documentation too: common ones include `private=True`, which forces a private message, and `notice=True`, which forces the reply to use NOTICE instead of PRIVMSG.

### 2.1.7 test.py

The easy way to test any plugin would be to start up a bot, load the plugin, and run all the commands a few times to verify that they work. But this takes time, and as a project grows larger, starts to be a tedious and error-prone process...

This is where automated testing comes in. Limnoria has a test harness built upon the [Python unittest library](#) that abstracts away all the dependencies of live testing (i.e. the IRC client and server) and allows you to cover your entire plugin’s functionality within a few seconds.

## How it works

Plugin test cases inherit from `supybot.test.PluginTestCase` or `supybot.test.ChannelPluginTestCase` and include [several methods](#) to interact with a simulated instance of the bot, in addition to the [standard assertion functions](#) provided by the unittest library.

Running the tests for a Limnoria plugin is done using the **supybot-test** command: i.e. `supybot-test /path/to/your/Plugin`

The structure of these test classes, as well as interactions with features like Limnoria’s config system are described in detail in the [Advanced Plugin Testing guide](#).

## Functional testing examples

For a command where we don't care about the exact output, the usual approach is to check that invocations raise or don't raise an error. For a command that generates a purely random output, this applies too since we can't predict what the result will be:

```
class RandomTestCase(PluginTestCase):
    # This tuple determines which plugins to load in the test case
    plugins = ('Random',)

    def testRandom(self):
        self.assertNotError('random')

        # This throws, because the command doesn't expect any arguments
        self.assertError('random abcdef')
```

However, this is less true if you pre-seed the RNG, as then you're guaranteed a repeatable result. The following snippet introduces `assertResponse(commandPlusArgs, expectedOutput)`, where `commandPlusArgs` is the full bot command including arguments, all as one string:

Alternatively, you can use `getMsg(command)` to fetch the output of a bot command as a string and reuse it:

Another common practice is to use regular expressions to match the output of a command:

---

**Note:** The `assertRegexp()` defined in Limnoria is *not* the same as `assertRegexp()` from the standard unittest library. The latter compares a regexp against a bare string, not the output of a bot command. (For historical reasons, we have this confusing name.)

---

### 2.1.8 Conclusion

You are now well prepared to write Limnoria plugins. A few words of wisdom:

- Read other people's plugins, especially the included plugins and ones by the core developers. We can't possibly document all the things that Limnoria can do, though we try our best.
- Hack new functionality into existing plugins first if writing a new plugin is too daunting.
- Come ask us questions in #limnoria on Libera. Going back to the first point above, the developers themselves can help you even more than the docs can (though we prefer you read the docs first).
- *Share your plugins with the world* and make Limnoria all that more attractive for other users so they will want to write their plugins for Limnoria as well.
- And of course, have fun!

## 2.2 Using `commands.wrap` to parse your command's arguments

### Contents

- *Using `commands.wrap` to parse your command's arguments*
  - *Introduction*
  - *Using `Wrap`*

- *Syntax Changes*
- *Customizing Wrap*
- *Converter List*
  - \* *Numbers and time*
  - \* *Channel*
  - \* *Words*
  - \* *Network*
  - \* *Users, nicks, and permissions*
  - \* *Matching*
  - \* *Other*
- *Contexts List*
  - \* *Options*
  - \* *Multiplicity*
  - \* *Other*
- *Final Word*

### 2.2.1 Introduction

To plugin developers for older (pre-0.80) versions of Supybot, one of the more annoying aspects of writing commands was handling the arguments that were passed in. In fact, many commands often had to duplicate parsing and verification code, resulting in lots of duplicated code for not a whole lot of action. So, instead of forcing plugin writers to come up with their own ways of cleaning it up, we wrote up the wrap function to handle all of it.

It allows a much simpler and more flexible way of checking things than before and it doesn't require that you know the bot internals to do things like check and see if a user exists, or check if a command name exists and whatnot.

If you are a plugin author this document is absolutely required reading, as it will massively ease the task of writing commands.

### 2.2.2 Using Wrap

First off, to get the wrap function, it is recommended (strongly) that you use the following import line:

```
from supybot.commands import *
```

This will allow you to access the wrap command (and it allows you to do it without the commands prefix). Note that this line is added to the imports of plugin templates generated by the supybot-plugin-create script.

Let's write a quickie command that uses wrap to get a feel for how it makes our lives better. Let's write a command that repeats a string of text a given number of times. So you could say "repeat 3 foo" and it would say "foofoofoo". Not a very useful command, but it will serve our purpose just fine. Here's how it would be done without wrap:

```
def repeat(self, irc, msg, args):
    """<num> <text>

    Repeats <text> <num> times.
```

(continues on next page)

(continued from previous page)

```
"""
(num, text) = privmsg.getArgs(args, required=2)
try:
    num = int(num)
except ValueError:
    raise callbacks.ArgumentError
irc.reply(num * text)
```

Note that all of the argument validation and parsing takes up 5 of the 6 lines (and you should have seen it before we had `privmsg.getArgs`!). Now, here's what our command will look like with `wrap` applied:

```
def repeat(self, irc, msg, args, num, text):
    """<num> <text>

    Repeats <text> <num> times.
    """
    irc.reply(text * num)
repeat = wrap(repeat, ['int', 'text'])
```

Pretty short, eh? With `wrap` all of the argument parsing and validation is handled for us and we get the arguments we want, formatted how we want them, and converted into whatever types we want them to be - all in one simple function call that is used to wrap the function! So now the code inside each command really deals with how to execute the command and not how to deal with the input.

So, now that you see the benefits of `wrap`, let's figure out what stuff we have to do to use it.

## 2.2.3 Syntax Changes

There are two syntax changes to the old style that are implemented. First, the definition of the command function must be changed. The basic syntax for the new definition is:

```
def commandname(self, irc, msg, args, <arg1>, <arg2>, ...):
```

Where `arg1` and `arg2` (up through as many as you want) are the variables that will store the parsed arguments. "Now where do these parsed arguments come from?" you ask. Well, that's where the second syntax change comes in. The second syntax change is the actual use of the `wrap` function itself to decorate our command names. The basic decoration syntax is:

```
commandname = wrap(commandname, [converter1, converter2, ...])
```

---

**Note:** This should go on the line immediately following the body of the command's definition, so it can easily be located (and it obviously must go after the command's definition so that `commandname` is defined).

---

Each of the converters in the above listing should be one of the converters in `commands.py` (I will describe each of them in detail later.) The converters are applied in order to the arguments given to the command, generally taking arguments off of the front of the argument list as they go. Note that each of the arguments is actually a string containing the NAME of the converter to use and not a reference to the actual converter itself. This way we can have converters with names like `int` and not have to worry about polluting the builtin namespace by overriding the builtin `int`.

As you will find out when you look through the list of converters below, some of the converters actually take arguments. The syntax for supplying them (since we aren't actually calling the converters, but simply specifying them), is to wrap the converter name and args list into a tuple. For example:

```
commandname = wrap(commandname, [(converterWithArgs, arg1, arg2),
                                converterWithoutArgs1, converterWithoutArgs2])
```

For the most part you won't need to use an argument with the converters you use either because the defaults are satisfactory or because it doesn't even take any.

## 2.2.4 Customizing Wrap

Converters alone are a pretty powerful tool, but for even more advanced (yet simpler!) argument handling you may want to use contexts. Contexts describe how the converters are applied to the arguments, while the converters themselves do the actual parsing and validation.

For example, one of the contexts is “optional”. By using this context, you're saying that a given argument is not required, and if the supplied converter doesn't find anything it likes, we should use some default. Yet another example is the “reverse” context. This context tells the supplied converter to look at the last argument and work backwards instead of the normal first-to-last way of looking at arguments.

So, that should give you a feel for the role that contexts play. They are not by any means necessary to use wrap. All of the stuff we've done to this point will work as-is. However, contexts let you do some very powerful things in very easy ways, and are a good thing to know how to use.

Now, how do you use them? Well, they are in the global namespace of `src/commands.py`, so your previous import line will import them all; you can call them just as you call wrap. In fact, the way you use them is you simply call the context function you want to use, with the converter (and its arguments) as arguments. It's quite simple. Here's an example:

```
commandname = wrap(commandname, [optional('int'), many('something')])
```

In this example, our command is looking for an optional integer argument first. Then, after that, any number of arguments which can be anything (as long as they are something, of course).

Do note, however, that the type of the arguments that are returned can be changed if you apply a context to it. So, `optional("int")` may very well return `None` as well as something that passes the “int” converter, because after all it's an optional argument and if it is `None`, that signifies that nothing was there. Also, for another example, `many("something")` doesn't return the same thing that just “something” would return, but rather a list of “something”s.

## 2.2.5 Converter List

Below is a list of all the available converters to use with wrap. If the converter accepts any arguments, they are listed after it and if they are optional, the default value is shown.

### Numbers and time

**expiry** Takes a number of seconds and adds it to the current time to create an expiration timestamp.

**id, kind="integer"** Returns something that looks like an integer ID number. Takes an optional “kind” argument for you to state what kind of ID you are looking for, though this doesn't affect the integrity-checking. Basically requires that the argument be an integer, does no other integrity-checking, and provides a nice error message with the kind in it.

**index** Basically (“int”, “index”), but with a twist. This will take a 1-based index and turn it into a 0-based index (which is more useful in code). It doesn't transform 0, and it maintains negative indices as is (note that it does allow them!).

**int, type="integer", p=None** Gets an integer. The “type” text can be used to customize the error message received when the argument is not an integer. “p” is an optional predicate to test the integer with. If p(i) fails (where i is the integer arg parsed out of the argument string), the arg will not be accepted.

**now** Simply returns the current timestamp as an arg, does not reference or modify the argument list.

**long, type="long"** Basically the same as int minus the predicate, except that it converts the argument to a long integer regardless of the size of the int.

**float, type="floating point number"** Basically the same as int minus the predicate, except that it converts the argument to a float.

**nonInt, type="non-integer value"** Accepts everything but integers, and returns them unchanged. The “type” value, as always, can be used to customize the error message that is displayed upon failure.

**positiveInt** Accepts only positive integers.

**nonNegativeInt** Accepts only non-negative integers.

### Channel

**channelDb** Sets the channel appropriately in order to get to the databases for that channel (handles whether or not a given channel uses channel-specific databases and whatnot).

**channel** Gets a channel to use the command in. If the channel isn’t supplied, uses the channel the message was sent in. If using a different channel, does sanity-checking to make sure the channel exists on the current IRC network.

**inChannel** Requires that the command be called from within any channel that the bot is currently in or with one of those channels used as an argument to the command.

**onlyInChannel** Requires that the command be called from within any channel that the bot is currently in.

**callerInGivenChannel** Takes the given argument as a channel and makes sure that the caller is in that channel.

**public** Requires that the command be sent in a channel instead of a private message.

**private** Requires that the command be sent in a private message instead of a channel.

**validChannel** Gets a channel argument once it makes sure it’s a valid channel.

### Words

**color** Accepts arguments that describe a text color code (e.g., “black”, “light blue”) and returns the mIRC color code for that color. (Note that many other IRC clients support the mIRC color code scheme, not just mIRC)

**letter** Looks for a single letter. (Technically, it looks for any one-element sequence).

**literal, literals, errmsg=None** Takes a required sequence or string (literals) and any argument that uniquely matches the starting substring of one of the literals is transformed into the full literal. For example, with ( "literal", ("bar", "baz", "qux") ), you’d get “bar” for “bar”, “baz” for “baz”, and “qux” for any of “q”, “qu”, or “qux”. “b” and “ba” would raise errors because they don’t uniquely identify one of the literals in the list. You can override errmsg to provide a specific (full) error message, otherwise the default argument error message is displayed.

**lowered** Returns the argument lowered (NOTE: it is lowered according to IRC conventions, which does strange mapping with some punctuation characters).

**to** Returns the string “to” if the arg is any form of “to” (case-insensitive).



## Network

**ip** Checks and makes sure the argument looks like a valid IP and then returns it.

**url** Checks for a valid URL.

**httpUrl** Checks for a valid HTTP URL.

## Users, nicks, and permissions

**haveOp, action="do that"** Simply requires that the bot have ops in the channel that the command is called in. The action parameter completes the error message: "I need to be opped to ...".

**nick** Checks that the arg is a valid nick on the current IRC server.

**seenNick** Checks that the arg is a nick that the bot has seen (NOTE: this is limited by the size of the history buffer that the bot has).

**nickInChannel** Requires that the argument be a nick that is in the current channel, and returns that nick.

**capability** Used to retrieve an argument that describes a capability.

**hostmask** Returns the hostmask of any provided nick or hostmask argument.

**banmask** Returns a generic banmask of the provided nick or hostmask argument.

**user** Requires that the caller be a registered user.

**otherUser** Returns the user specified by the username or hostmask in the argument.

**owner** Requires that the command caller has the "owner" capability.

**admin** Requires that the command caller has the "admin" capability.

**checkCapability, capability** Checks to make sure that the caller has the specified capability.

**checkChannelCapability, capability** Checks to make sure that the caller has the specified capability on the channel the command is called in.

## Matching

**anything** Returns anything as is.

**something, errorMsg=None, p=None** Takes anything but the empty string. errorMsg can be used to customize the error message. p is any predicate function that can be used to test the validity of the input.

**somethingWithoutSpaces** Same as something, only with the exception of disallowing spaces of course.

**matches, regexp, errmsg** Searches the args with the given regexp and returns the matches. If no match is found, errmsg is given.

**regexpMatcher** Gets a matching regexp argument (m// or //).

**glob** Gets a glob string. Basically, if there are no wildcards (\*, ?) in the argument, returns \*string\*, making a glob string that matches anything containing the given argument.

**regexpReplacer** Gets a replacing regexp argument (s//).

### Other

**networkIrc, errorIfNoMatch=False** Returns the IRC object of the specified IRC network. If one isn't specified, the IRC object of the IRC network the command was called on is returned.

**plugin, require=True** Returns the plugin specified by the arg or None. If require is True, an error is raised if the plugin cannot be retrieved.

**boolean** Converts the text string to a boolean value. Acceptable true values are: "1", "true", "on", "enable", or "enabled" (case-insensitive). Acceptable false values are: "0", "false", "off", "disable", or "disabled" (case-insensitive).

**filename** Used to get a filename argument.

**commandName** Returns the canonical command name version of the given string (ie, the string is lowercased and dashes and underscores are removed).

**text** Takes the rest of the arguments as one big string. Note that this differs from the "anything" context in that it clobbers the arg string when it's done. Using any converters after this is most likely incorrect.

### 2.2.6 Contexts List

What contexts are available for me to use?

The list of available contexts is below. Unless specified otherwise, it can be assumed that the type returned by the context itself matches the type of the converter it is applied to.

### Options

**optional** Look for an argument that satisfies the supplied converter, but if it's not the type I'm expecting or there are no arguments for us to check, then use the default value. Will return the converted argument as is or None.

**additional** Look for an argument that satisfies the supplied converter, making sure that it's the right type. If there aren't any arguments to check, then use the default value. Will return the converted argument as is or None.

**first** Tries each of the supplied converters in order and returns the result of the first successfully applied converter.

### Multiplicity

**any** Looks for any number of arguments matching the supplied converter. Will return a sequence of converted arguments or None.

**many** Looks for multiple arguments matching the supplied converter. Expects at least one to work, otherwise it will fail. Will return the sequence of converted arguments.

**commalist** Looks for a comma separated list of arguments that match the supplied converter. Returns a list of the successfully converted arguments. If any of the arguments fail, this whole context fails.

### Other

**rest** Treat the rest of the arguments as one big string, and then convert. If the conversion is unsuccessful, restores the arguments.

**getopts** Handles -option style arguments. Each option should be a key in a dictionary that maps to the name of the converter that is to be used on that argument. To make the option take no argument, use "" as the converter name in the dictionary. For no conversion, use None as the converter name in the dictionary.

**reverse** Reverse the argument list, apply the converters, and then reverse the argument list back.

## 2.2.7 Final Word

Now that you know how to use wrap, and you have a list of converters and contexts you can use, your task of writing clean, simple, and safe plugin code should become much easier. Enjoy!

## 2.3 Style Guidelines

**Note:** Code not following these style guidelines fastidiously is likely not to be accepted into the Limnoria core.

- Read [PEP 8](#) (Guido's Style Guide) and know that we use almost all the same style guidelines. - We use a maximum of 79 characters per line and 4 spaces per indentation level - Exception: method and function names generally use camelCase for consistency with existing code.
- Raw strings (`r' '` or `r" "`) should be used for regular expressions.
- Unless absolutely required by some external force, imports should be ordered by the string length of the module imported. I just think it looks prettier.
- Database filenames should generally begin with the name of the plugin and the extension should be 'db'. `plugins.DBHandler` does this already.
- Whenever creating a file descriptor or socket, keep a reference around and be sure to close it. There should be no code like this:

```
s = urllib.request.urlopen('url').read()
```

Instead, do this:

```
fd = urllib.request.urlopen('url')
try:
    s = fd.read()
finally:
    fd.close()
```

This is to be sure the bot doesn't leak file descriptors.

- All plugin files should include a docstring describing what the plugin does. This docstring will be returned when the user is configuring the plugin.
- All plugin classes should also include a docstring describing how to do things with the plugin; this docstring will be returned when the user requests help on a plugin name.
- Method docstrings in classes deriving from `callbacks.Privmsg` should include an argument list as their first line, and after that a blank line followed by a longer description of what the command does. The argument list is used by the `syntax` command, and the longer description is used by the `help` command.
- Whenever joining more than two strings, use f-strings or string interpolation, not addition:

```
s = x + y + z # Bad.
s = '%s%s%s' % (x, y, z) # Good.
s = ''.join([x, y, z]) # Better, but not as general.
s = f'{x}{y}{z}' # Best.
```

- When writing strings that have formatting characters in them, don't use anything but `%s` unless you absolutely must. Avoid `%d` in particular because it's not as general and is likely to throw type errors if you make a mistake.

- As a corollary to the above, note that sometimes `%f` is used, but only when floats need to be formatted, e.g., `%.2f`.
- Use the log module to its fullest; when you need to print some values to debug, use `self.log.debug` to do so, and leave those statements in the code (commented out) so they can later be re-enabled. Remember that once code is buggy, it tends to have more bugs, and you'll probably need those print statements again.
- While on the topic of logs, note that we do not use `%` (i.e., `str.__mod__`) with logged strings; we simply pass the format parameters as additional arguments. The reason is simple: the logging module supports it, and it's cleaner (fewer tokens/glyphs) to read.
- While still on the topic of logs, it's also important to pick the appropriate log level for given information.
  - **DEBUG**: Appropriate to tell a programmer *how* we're doing something (i.e., debugging `printf`s, basically). If you're trying to figure out why your code doesn't work, **DEBUG** is the new `printf` – use that, and leave the statements in your code.
  - **INFO**: Appropriate to tell a user *what* we're doing, when what we're doing isn't important for the user to pay attention to. A user who likes to keep up with things should enjoy watching our logging at the **INFO** level; it shouldn't be too low-level, but it should give enough information that it keeps them relatively interested at peak times.
  - **WARNING**: Appropriate to tell a user when we're doing something that they really ought to pay attention to. Users should see **WARNING** and think, "Hmm, should I tell the Limnoria developers about this?" Later, they should decide not to, but it should give the user a moment to pause and think about what's actually happening with their bot.
  - **ERROR**: Appropriate to tell a user when something has gone wrong. Uncaught exceptions are **ERROR**s. Conditions that we absolutely want to hear about should be errors. Things that should *scare* the user should be errors.
  - **CRITICAL**: Not really appropriate. I can think of no absolutely critical issue yet encountered in Limnoria; the only possible thing I can imagine is to notify the user that the partition on which Limnoria is running has filled up. That would be a **CRITICAL** condition, but it would also be hard to log :)
- All plugins should have test cases written for them. Even if it doesn't actually test anything but just exists, it's good to have the test there so there's a place to add more tests later (and so we can be sure that all plugins are adequately documented; `PluginTestCase` checks that every command has documentation)
- SQL table names should be all-lowercase and include underscores to separate words. This is because SQL itself is case-insensitive. This doesn't change, however the fact that variable/member names should be camel case.
- SQL statements in code should put SQL words in ALL CAPS:

```
"""SELECT quote FROM quotes ORDER BY random() LIMIT 1"""
```

This makes SQL significantly easier to read.

- Common variable names
  - `L` => an arbitrary list.
  - `t` => an arbitrary tuple.
  - `x` => an arbitrary float.
  - `s` => an arbitrary string.
  - `f` => an arbitrary function.
  - `p` => an arbitrary predicate.
  - `i,n` => an arbitrary integer.
  - `cb` => an arbitrary callback.

- db => a database handle.
- fd => a file-like object.
- msg => an ircmsgs.IrcMsg object.
- irc => an irclib.Irc object (or proxy)
- nick => a string that is an IRC nick.
- channel => a string that is an IRC channel.
- hostmask => a string that is a user's IRC prefix.

When the semantic functionality (that is, the “meaning” of a variable is obvious from context), one of these names should be used. This just makes it easier for people reading our code to know what a variable represents without scouring the surrounding code.

- Multiple variable assignments should always be surrounded with parentheses – i.e., if you're using the partition function, then your assignment statement should look like:

```
(good, bad) = partition(p, L)
```

The parentheses make it obvious that you're doing a multiple assignment, and that's important because I hate reading code and wondering where a variable came from.

## 2.4 Advanced Plugin Config

This tutorial covers some of the more advanced plugin config features available to Supybot plugin authors.

### 2.4.1 What's This Tutorial For?

Brief overview of what this tutorial covers and the target audience.

Want to know the crazy advanced features available to you, the Supybot plugin author? Well, this is the tutorial for you. This article assumes you've read the Supybot plugin author tutorial since all the basics of plugin config are handled there first.

In this tutorial we'll cover:

- Creating config variable groups and config variables underneath those groups.
- The built-in config variable types (“registry types”) for use with config variables
- Creating custom registry types to handle config variable values more effectively
- Using the configure function for interactive configuration in supybot-wizard

### 2.4.2 Using Config Groups

A brief overview of how to use config groups to organize config variables

## Supybot's Hierarchical Configuration

Supybot's configuration is inherently hierarchical, as you've probably already figured out in your use of the bot. Naturally, it makes sense to allow plugin authors to create their own hierarchies to organize their configuration variables for plugins that have a lot of plugin options. If you've taken a look at the plugins that Supybot comes with, you've probably noticed that several of them take advantage of this. In this section of this tutorial we'll go over how to make your own config hierarchy for your plugin.

Here's the brilliant part about Supybot config values which makes hierarchical structuring all that much easier - values are groups. That is, any config value you may already defined in your plugins can already be treated as a group, you simply need to know how to add items to that group.

Now, if you want to just create a group that doesn't have an inherent value you can do that as well, but you'd be surprised at how rarely you have to do that. In fact if you look at most of the plugins that Supybot comes with, you'll only find that we do this in a handful of spots yet we use the "values as groups" feature quite a bit.

### Creating a Config Group

As stated before, config variables themselves are groups, so you can create a group simply by creating a configuration variable:

```
conf.registerGlobalValue(WorldDom, 'globalWorldDominationRequires',
    registry.String('', ""Determines the capability required to access the
                    world domination commands in this plugin.""))
```

As you probably know by now this creates the config variable `supybot.plugins.WorldDom.globalWorldDominationRequires` which you can access/set using the Config plugin directly on the running bot. What you may not have known prior to this tutorial is that that variable is also a group. Specifically, it is now the `WorldDom.globalWorldDominationRequires` group, and we can add config variables to it! Unfortunately, this particular bit of configuration doesn't really require anything underneath it, so let's create a new group which does using the "create only a group, not a value" command.

Let's create a configurable list of targets for different types of attacks (land, sea, air, etc.). We'll call the group `attackTargets`. Here's how you create just a config group alone with no value assigned:

```
conf.registerGroup(WorldDom, 'attackTargets')
```

The first argument is just the group under which you want to create your new group (and we got `WorldDom` from `conf.registerPlugin` which was in our boilerplate code from the plugin creation wizard). The second argument is, of course, the group name. So now we have `WorldDom.attackTargets` (or, fully, `supybot.plugins.WorldDom.attackTargets`).

### Adding Values to a Group

Actually, you've already done this several times, just never to a custom group of your own. You've always added config values to your plugin's config group. With that in mind, the only slight modification needed is to simply point to the new group:

```
conf.registerGlobalValue(WorldDom.attackTargets, 'air',
    registry.SpaceSeparatedListOfStrings('', ""Contains the list of air
                                         targets.""))
```

And now we have a nice list of air targets! You'll notice that the first argument is `WorldDom.attackTargets`, our new group. Make sure that the `conf.registerGroup` call is made before this one or else you'll get a nasty `AttributeError`.

## Variations

### Channel-specific values

A very handy feature is channel-specific variables, which allows bot administrators to set a global value (as for non-channel-specific values AND another value for specific channels).

The syntax is pretty much like the previous one, except we use *registerChannelValue* instead of *registerGlobalValue*:

```
conf.registerChannelValue(WorldDom.attackTargets, 'air',
    registry.SpaceSeparatedListOfStrings(' ', '"Contains the list of air
                                           targets."'))
```

### Private values

Variable type also take an optional argument, for setting a configuration variable to private (useful for passwords, authentication tokens, api keys, ...):

```
conf.registerChannelValue(WorldDom.attackTargets, 'air',
    registry.SpaceSeparatedListOfStrings(' ', '"Contains the list of air
                                           targets."', private=True))
```

### Accessing the configuration registry

Of course, you can access the variables in your plugins.

If it is a variable created by your plugin, you can do it like this (if the configuration variable's name is *air*):

```
self.registryValue('air')
```

and it will return data of the right type (in this case, a list of string, as we declared it above as a *registry.SpaceSeparatedListOfStrings*).

If it is a channel-specific variable, you can get the value on *#channel* and *network* like this (if the variable is not defined on this channel, it defaults to the global one):

```
self.registryValue('air', '#channel', 'network')
```

**Note:** You will typically obtain the current channel name using the *channel* converter (in commands with a *<channel>* argument) or *msg.channel* (in other methods); and the network name with *irc.network*.

You can also set configuration variables (either globally or for a single channel):

```
self.setRegistryValue('air', value=['foo', 'bar'])
self.setRegistryValue('air', value=['foo', 'bar'],
    channel=channel, network=network)
```

You can also access other configuration variables (or your own if you want) via the *supybot.conf* module:

```
conf.supybot.plugins.WorldDom.air()
conf.supybot.plugins.WorldDom.get('air')()
conf.supybot.plugins.WorldDom.air.get('network').get('#channel')()
```

(continues on next page)

(continued from previous page)

```
conf.supybot.plugins.WorldDom.air.setValue(['foo'])
conf.supybot.plugins.WorldDom.air.get('network').get('#channel').setValue(['foo'])
```

**Warning:** Before version 2019.10.22, Limnoria (and Supybot) did not support network-specific configuration variables. If you want to support these versions, you must drop the *network* argument, and access the configuration variables like this:

```
self.registryValue('air', '#channel', 'network')
self.setRegistryValue('air', value=['foo', 'bar'],
                      channel=channel)
conf.supybot.plugins.WorldDom.air.get('#channel')()
conf.supybot.plugins.WorldDom.air.get('#channel').setValue(['foo'])
```

This will also work in recent versions of Limnoria, but will prevent users from setting different values for each network.

## 2.4.3 The Built-in Registry Types

A rundown of all of the built-in registry types available for use with config variables.

The “registry” module defines the following config variable types for your use (I’ll include the ‘registry.’ on each one since that’s how you’ll refer to it in code most often). Most of them are fairly self-explanatory, so excuse the boring descriptions:

- `registry.Boolean` - A simple true or false value. Also accepts the following for true: “true”, “on” “enable”, “enabled”, “1”, and the following for false: “false”, “off”, “disable”, “disabled”, “0”,
- `registry.Integer` - Accepts any integer value, positive or negative.
- `registry.NonNegativeInteger` - Will hold any non-negative integer value.
- `registry.PositiveInteger` - Same as above, except that it doesn’t accept 0 as a value.
- `registry.Float` - Accepts any floating point number.
- `registry.PositiveFloat` - Accepts any positive floating point number.
- `registry.Probability` - Accepts any floating point number between 0 and 1 (inclusive, meaning 0 and 1 are also valid).
- `registry.String` - Accepts any string that is not a valid Python command
- `registry.NormalizedString` - Accepts any string (with the same exception above) but will normalize sequential whitespace to a single space..
- `registry.StringSurroundedBySpaces` - Accepts any string but assures that it has a space preceding and following it. Useful for configuring a string that goes in the middle of a response.
- `registry.StringWithSpaceOnRight` - Also accepts any string but assures that it has a space after it. Useful for configuring a string that begins a response.
- `registry.Regexp` - Accepts only valid (Perl or Python) regular expressions
- `registry.SpaceSeparatedListOfStrings` - Accepts a space-separated list of strings.

There are a few other built-in registry types that are available but are not usable in their current state, only by creating custom registry types, which we’ll go over in the next section.



## 2.4.4 Custom Registry Types

How to create and use your own custom registry types for use in customizing plugin config variables.

### Why Create Custom Registry Types?

For most configuration, the provided types in the registry module are sufficient. However, for some configuration variables it's not only convenient to use custom registry types, it's actually recommended. Customizing registry types allows for tighter restrictions on the values that get set and for greater error-checking than is possible with the provided types.

### What Defines a Registry Type?

First and foremost, it needs to subclass one of the existing registry types from the registry module, whether it be one of the ones in the previous section or one of the other classes in registry specifically designed to be subclassed.

Also it defines a number of other nice things: a custom error message for your type, customized value-setting (transforming the data you get into something else if wanted), etc.

### Creating Your First Custom Registry Type

As stated above, priority number one is that you subclass one of the types in the registry module. Basically, you just subclass one of those and then customize whatever you want. Then you can use it all you want in your own plugins. We'll do a quick example to demonstrate.

We already have `registry.Integer` and `registry.PositiveInteger`, but let's say we want to accept only negative integers. We can create our own `NegativeInteger` registry type like so:

```
class NegativeInteger(registry.Integer):
    """Value must be a negative integer."""
    def setValue(self, v):
        if v >= 0:
            self.error()
        registry.Integer.setValue(self, v)
```

All we need to do is define a new error message for our custom registry type (specified by the docstring for the class), and customize the `setValue` function. Note that all you have to do when you want to signify that you've gotten an invalid value is to call `self.error()`. Finally, we call the parent class's `setValue` to actually set the value.

### What Else Can I Customize?

Well, the error string and the `setValue` function are the most useful things that are available for customization, but there are other things. For examples, look at the actual built-in registry types defined in `registry.py` (in the `src` directory distributed with the bot).

### What Subclasses Can I Use?

Chances are one of the built-in types in the previous section will be sufficient, but there are a few others of note which deserve mention:

- `registry.Value` - Provides all the core functionality of registry types (including acting as a group for other config variables to reside underneath), but nothing more.

- `registry.OnlySomeStrings` - Allows you to specify only a certain set of strings as valid values. Simply override `validStrings` in the inheriting class and you're ready to go.
- `registry.SeparatedListOf` - The generic class which is the parent class to `registry.SpaceSeparatedListOfStrings`. Allows you to customize four things: the type of sequence it is (list, set, tuple, etc.), what each item must be (String, Boolean, etc.), what separates each item in the sequence (using custom splitter/joiner functions), and whether or not the sequence is to be sorted. Look at the definitions of `registry.SpaceSeparatedListOfStrings` and `registry.CommaSeparatedListOfStrings` at the bottom of `registry.py` for more information. Also, there will be an example using this in the section below.

## Using My Custom Registry Type

Using your new registry type is relatively straightforward. Instead of using whatever registry built-in you might have used before, now use your own custom class. Let's say we define a registry type to handle a comma-separated list of probabilities:

```
class CommaSeparatedListOfProbabilities(registry.SeparatedListOf):
    Value = registry.Probability
    def splitter(self, s):
        return re.split(r'\s*,\s*', s)
    joiner = ', '.join
```

Now, to use that type we simply have to specify it whenever we create a config variable using it:

```
conf.registerGlobalValue(SomePlugin, 'someConfVar',
    CommaSeparatedListOfProbabilities('0.0, 1.0', """Holds the list of
probabilities for whatever."""))
```

Note that we initialize it just the same as we do any other registry type, with two arguments: the default value, and then the description of the config variable.

## 2.4.5 Using ‘configure’ for supybot-wizard support

How to use ‘configure’ effectively using the functions from ‘supybot.questions’

In the original Supybot plugin author tutorial you'll note that we gloss over the configure portion of the `config.py` file for the sake of keeping the tutorial to a reasonable length. Well, now we're going to cover it in more detail.

The `supybot.questions` module is a nice little module coded specifically to help clean up the configure section of every plugin's `config.py`. The boilerplate `config.py` code imports the four most useful functions from that module:

- “expect” is a very general prompting mechanism which can specify certain inputs that it will accept and also specify a default response. It takes the following arguments:
  - prompt: The text to be displayed
  - possibilities: The list of possible responses (can be the empty list, [])
  - default (optional): Defaults to None. Specifies the default value to use if the user enters in no input.
  - acceptEmpty (optional): Defaults to False. Specifies whether or not to accept no input as an answer.
- “anything” is basically a special case of expect which takes anything (including no input) and has no default value specified. It takes only one argument:
  - prompt: The text to be displayed
- “something” is also a special case of expect, requiring some input and allowing an optional default. It takes the following arguments:

- prompt: The text to be displayed
- default (optional): Defaults to None. The default value to use if the user doesn’t input anything.
- “yn” is for “yes or no” questions and basically forces the user to input a “y” for yes, or “n” for no. It takes the following arguments:
  - prompt: The text to be displayed
  - default (optional): Defaults to None. Default value to use if the user doesn’t input anything.

All of these functions, with the exception of “yn”, return whatever string results as the answer whether it be input from the user or specified as the default when the user inputs nothing. The “yn” function returns True for “yes” answers and False for “no” answers.

For the most part, the latter three should be sufficient, but we expose expect to anyone who needs a more specialized configuration.

Let’s go through a quick example configure that covers all four of these functions. First I’ll give you the code, and then we’ll go through it, discussing each usage of a supybot.questions function just to make sure you realize what the code is actually doing. Here it is:

```
def configure(advanced):
    # This will be called by supybot to configure this module.  advanced is
    # a bool that specifies whether the user identified himself as an advanced
    # user or not.  You should effect your configuration by manipulating the
    # registry as appropriate.
    from supybot.questions import expect, anything, something, yn
    WorldDom = conf.registerPlugin('WorldDom', True)
    if yn("""The WorldDom plugin allows for total world domination
        with simple commands.  Would you like these commands to
        be enabled for everyone?""", default=False):
        WorldDom.globalWorldDominationRequires.setValue("")
    else:
        cap = something("""What capability would you like to require for
            this command to be used?""", default="Admin")
        WorldDom.globalWorldDominationRequires.setValue(cap)
    dir = expect("""What direction would you like to attack from in
        your quest for world domination?""",
        ["north", "south", "east", "west", "ABOVE"],
        default="ABOVE")
    WorldDom.attackDirection.setValue(dir)
```

As you can see, this is the WorldDom plugin, which I am currently working on. The first thing our configure function checks is to see whether or not the bot owner would like the world domination commands in this plugin to be available to everyone. If they say yes, we set the globalWorldDominationRequires configuration variable to the empty string, signifying that no specific capabilities are necessary. If they say no, we prompt them for a specific capability to check for, defaulting to the “Admin” capability. Here they can create their own custom capability to grant to folks which this plugin will check for if they want, but luckily for the bot owner they don’t really have to do this since Supybot’s capabilities system can be flexed to take care of this.

Lastly, we check to find out what direction they want to attack from as they venture towards world domination. I prefer “death from above!”, so I made that the default response, but the more boring cardinal directions are available as choices as well.

## 2.4.6 Configuration hooks

---

**Note:** This feature is specific to Limnoria and not available in stock Supybot or Gribble.

---

It is possible to get a function called when a configuration variable is changed. While this is usually not useful (you get the value whenever you need it), some plugins do use it, for instance for caching results or for pre-fetching data.

Let's say you want to write a plugin that prints *nick changed* in the logs when *supybot.nick* is edited. You can do it like this:

```
class LogNickChange(callbacks.Plugin):
    """Some useless plugin."""

    def __init__(self, irc):
        super().__init__(irc)
        conf.supybot.nick.addCallback(self._configCallback)

    def _configCallback(self, name=None):
        self.log.info('nick changed')
```

As not all Supybot versions support it (yet), it can be a good idea to show a warning instead of crashing on those versions:

```
class LogNickChange(callbacks.Plugin):
    """Some useless plugin."""

    def __init__(self, irc):
        super().__init__(irc)
        try:
            conf.supybot.nick.addCallback(self._configCallback)
        except registry.NonExistentRegistryEntry:
            self.log.error('Your version of Supybot is not compatible '
                           'with configuration hooks, but this plugin '
                           'requires them to work.')

    def _configCallback(self, name=None):
        self.log.info('nick changed')
```

---

**Note:** For the moment, the *name* parameter is never given when the callback is called. However, in the future, it will be set to the name of the variable that has been changed (useful if you want to use the same callback for multiple variable), so it is better to allow this parameter.

---

## 2.5 Advanced Plugin Testing

The complete guide to writing tests for your plugins.

### 2.5.1 Why Write Tests?

Why should I write tests for my plugin? Here's why.

For those of you asking “Why should I write tests for my plugin? I tried it out, and it works!”, read on. For those of you who already realize that Testing is Good (TM), skip to the next section.

Here are a few quick reasons why to test your Supybot plugins.

- When/if we rewrite or change certain features in Supybot, tests make sure your plugin will work with these changes. It's much easier to run `supybot-test MyPlugin` after upgrading the code and before even reloading the bot with the new code than it is to load the bot with new code and then load the plugin only to realize certain things don't work. You may even ultimately decide you want to stick with an older version for a while as you patch your custom plugin. This way you don't have to rush a patch while restless users complain since you're now using a newer version that doesn't have the plugin they really like.
- Running the automated tests takes a few seconds, testing plugins in IRC on a live bot generally takes quite a bit longer. We make it so that writing tests generally doesn't take much time, so a small initial investment adds up to lots of long-term gains.
- If you want your plugin to be included in any of our releases (the core Supybot if you think it's worthy, or our `supybot-plugins` package), it has to have tests. Period.

For a bigger list of why to write unit tests, check out this article:

<http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html>

and also check out what the Extreme Programming folks have to say about unit tests:

<http://www.extremeprogramming.org/rules/unittests.html>

## 2.5.2 Plugin Tests

How to write tests for commands in your plugins.

### Introduction

This tutorial assumes you've read through the plugin author tutorial, and that you used `supybot-plugin-create` to create your plugin (as everyone should). So, you should already have all the necessary imports and all that boilerplate stuff in `test.py` already, and you have already seen what a basic plugin test looks like from the plugin author tutorial. Now we'll go into more depth about what plugin tests are available to Supybot plugin authors.

### Plugin Test Case Classes

Supybot comes with two plugin test case classes, `PluginTestCase` and `ChannelPluginTestCase`. The former is used when it doesn't matter whether or not the commands are issued in a channel, and the latter is used for when it does. For the most part their API is the same, so unless there's a distinction between the two we'll treat them as one and the same when discussing their functionality.

### The Most Basic Plugin Test Case

At the most basic level, a plugin test case requires three things:

- the class declaration (subclassing `PluginTestCase` or `ChannelPluginTestCase`)
- a list of plugins that need to be loaded for these tests (does not include `Owner`, `Misc`, or `Config`, those are always automatically loaded) - often this is just the name of the plugin that you are writing tests for
- some test methods

Here's what the most basic plugin test case class looks like (for a plugin named `MyPlugin`):

```
class MyPluginTestCase(PluginTestCase):
    plugins = ('MyPlugin',)

    def testSomething(self):
        # assertions and such go here
```

Your plugin test case should be named `TestCase` as you see above, though it doesn't necessarily have to be named that way (supybot-plugin-create puts that in place for you anyway). As you can see we elected to subclass `PluginTestCase` because this hypothetical plugin apparently doesn't do anything channel-specific.

As you probably noticed, the `plugins` attribute of the class is where the list of necessary plugins goes, and in this case just contains the plugin that we are testing. This will be the case for probably the majority of plugins. A lot of the time test writers will use a bot function that performs some function that they don't want to write code for and they will just use command nesting to feed the bot what they need by using that plugin's functionality. If you choose to do this, only do so with core bot plugins as this makes distribution of your plugin simpler. After all, we want people to be able to run your plugin tests without having to have all of your plugins!

One last thing to note before moving along is that each of the test methods should describe what they are testing. If you want to test that your plugin only responds to registered users, don't be afraid to name your test method `testOnlyRespondingToRegisteredUsers` or `testNotRespondingToUnregisteredUsers`. You may have noticed some rather long and seemingly unwieldy test method names in our code, but that's okay because they help us know exactly what's failing when we run our tests. With an ambiguously named test method we may have to crack open `test.py` after running the tests just to see what it is that failed. For this reason you should also test only one thing per test method. Don't write a test method named `testFoobarAndBaz`. Just write two test methods, `testFoobar` and `testBaz`. Also, it is important to note that test methods must begin with `test` and that any method within the class that does begin with `test` will be run as a test by the `supybot-test` program. If you want to write utility functions in your test class that's fine, but don't name them something that begins with `test` or they will be executed as tests.

## Including Extra Setup

Some tests you write may require a little bit of setup. For the most part it's okay just to include that in the individual test method itself, but if you're duplicating a lot of setup code across all or most of your test methods it's best to use the `setUp` method to perform whatever needs to be done prior to each test method.

The `setUp` method is inherited from the whichever plugin test case class you chose for your tests, and you can add whatever functionality you want to it. Note the important distinction, however: you should be adding to it and not overriding it. Just define `setUp` in your own plugin test case class and it will be run before all the test methods are invoked.

Let's do a quick example of one. Let's write a `setUp` method which registers a test user for our test bot:

```
def setUp(self):
    ChannelPluginTestCase.setUp(self) # important!!
    # Create a valid user to use
    self.prefix = 'foo!bar@baz'
    self.feedMsg('register tester moo', to=self.nick, frm=self.prefix)
    m = self.getMsg(' ') # Response to registration.
```

Now notice how the first line calls the parent class's `setUp` method first? This must be done first. Otherwise several problems are likely to arise. For one, you wouldn't have an `irc` object at `self.irc` that we use later on nor would `self.nick` be set.

As for the rest of the method, you'll notice a few things that are available to the plugin test author. `self.prefix` refers to the hostmask of the hypothetical test user which will be "talking" to the bot, issuing commands. We set it to some generically fake hostmask, and then we use `feedMsg` to send a private message (using the bot's nick, accessible via `self.nick`) to the bot registering the username "tester" with the password "moo". We have to do it this way (rather than

what you'll find out is the standard way of issuing commands to the bot in test cases a little later) because registration must be done in private. And lastly, since `feedMsg` doesn't dequeue any messages from the bot after being fed a message, we perform a `getMsg` to get the response. You're not expected to know all this yet, but do take note of it since using these methods in test-writing is not uncommon. These utility methods as well as all of the available assertions are covered in the next section.

So, now in any of the test methods we write, we'll be able to count on the fact that there will be a registered user "tester" with a password of "moo", and since we changed our prefix by altering `self.prefix` and registered after doing so, we are now identified as this user for all messages we send unless we specify that they are coming from some other prefix.

## The Opposite of Setting-up: Tearing Down

If you did some things in your `setUp` that you want to clean up after, then this code belongs in the `tearDown` method of your test case class. It's essentially the same as `setUp` except that you probably want to wait to invoke the parent class's `tearDown` until after you've done all of your tearing down. But do note that you do still have to invoke the parent class's `tearDown` method if you decide to add in your own tear-down stuff.

## Setting Config Variables for Testing

Before we delve into all of the fun assertions we can use in our test methods it's worth noting that each plugin test case can set custom values for any Supybot config variable they want rather easily. Much like how we can simply list the plugins we want loaded for our tests in the `plugins` attribute of our test case class, we can set config variables by creating a mapping of variables to values with the `config` attribute.

So if, for example, we wanted to disable nested commands within our plugin testing for some reason, we could just do this:

```
class MyPluginTestCase(PluginTestCase):
    config = {'supybot.commands.nested': False}

    def testThisThing(self):
        # stuff
```

And now you can be assured that `supybot.commands.nested` is going to be off for all of your test methods in this test case class.

## Temporarily setting a configuration variable

Sometimes we want to change a configuration variable only in a test (or in a part of a test), and keep the original value for other tests. The historical way to do it is:

```
import supybot.conf as conf

class MyPluginTestCase(PluginTestCase):
    def testThisThing(self):
        original_value = conf.supybot.commands.nested()
        conf.supybot.commands.nested.setValue(False)
        try:
            # stuff
        finally:
            conf.supybot.commands.nested.setValue(original_value)
```

But there is a more compact syntax, using context managers:

```
import supybot.conf as conf

class MyPluginTestCase(PluginTestCase):
    def testThisThing(self):
        with conf.supybot.commands.nested.context(False):
            # stuff
```

### 2.5.3 Plugin Test Methods

The full list of test methods and how to use them.

#### Introduction

You know how to make plugin test case classes and you know how to do just about everything with them except to actually test stuff. Well, listed below are all of the assertions used in tests. If you're unfamiliar with what an assertion is in code testing, it is basically a requirement of something that must be true in order for that test to pass. It's a necessary condition. If any assertion within a test method fails the entire test method fails and it goes on to the next one.

#### Assertions

All of these are methods of the plugin test classes themselves and hence are accessed by using `self.assertWhatever` in your test methods. These are sorted in order of relative usefulness.

**assertResponse(query, expectedResponse)** Feeds query to the bot as a message and checks to make sure the response is `expectedResponse`. The test fails if they do not match (note that prefixed nicks in the response do not need to be included in the `expectedResponse`).

**assertError(query)** Feeds query to the bot and expects an error in return. Fails if the bot doesn't return an error.

**assertNotError(query)** The opposite of `assertError`. It doesn't matter what the response to query is, as long as it isn't an error. If it is not an error, this test passes, otherwise it fails.

**assertRegex(query, regexp, flags=re.I)** Feeds query to the bot and expects something matching the regexp (no `m//` required) in regexp with the supplied flags. Fails if the regexp does not match the bot's response.

**assertNotRegex(query, regexp, flags=re.I)** The opposite of `assertRegex`. Fails if the bot's output matches regexp with the supplied flags.

**assertHelp(query)** Expects query to return the help for that command. Fails if the command help is not triggered.

**assertAction(query, expectedResponse=None)** Feeds query to the bot and expects an action in response, specifically `expectedResponse` if it is supplied. Otherwise, the test passes for any action response.

**assertActionRegex(query, regexp, flags=re.I)** Basically like `assertRegex` but carries the extra requirement that the response must be an action or the test will fail.

#### Utilities

**feedMsg(query, to=None, frm=None)** Simply feeds query to whoever is specified in `to` or to the bot itself if no one is specified. Can also optionally specify the hostmask of the sender with the `frm` keyword. Does not actually perform any assertions.

**getMsg(query)** Feeds query to the bot and gets the response.



## 2.5.4 Other Tests

If you had to write helper code for a plugin and want to test it, here's how.

Previously we've only discussed how to test stuff in the plugin that is intended for IRC. Well, we realize that some Supybot plugins will require utility code that doesn't necessarily require all of the overhead of setting up IRC stuff, and so we provide a more lightweight test case class, `SupyTestCase`, which is a very very light wrapper around `unittest.TestCase` (from the standard `unittest` module) that basically just provides a little extra logging. This test case class is what you should use for writing those test cases which test things that are independent of IRC.

For example, in the `MoobotFactoids` plugin there is a large chunk of utility code dedicating to parsing out random choices within a factoid using a class called `OptionList`. So, we wrote the `OptionListTestCase` as a `SupyTestCase` for the `MoobotFactoids` plugin. The setup for test methods is basically the same as before, only you don't have to define plugins since this is independent of IRC.

You still have the choice of using `setUp` and `tearDown` if you wish, since those are inherited from `unittest.TestCase`. But, the same rules about calling the `setUp` or `tearDown` method from the parent class still apply.

With all this in hand, now you can write great tests for your Supybot plugins!

## 2.6 Distributing plugins

Now that you wrote a plugin, you may want to share it with other people. There are many way to do this, and we are going to present some of them below.

### 2.6.1 Via a VCS repository

If you already use a VCS (such as Git, Mercurial, or Subversion), giving access to your repository is the easiest way for you to share your plugins. This method can also easily be adapted to distribute tarballs.

It will however require a little more work for your users.

There are two ways to do it:

#### With a single plugin per repository

By initializing a repository in each folder created by *supybot-plugin-wizard* (eg. with *git init*). This allows you to keep each plugin's history independent.

First, users should install a client for your VCS, and dependencies of your plugin, if any.

Users can then download and install your plugin in a single command:

```
cd runbot/plugins/
git clone https://example.org/~jdoe/YourPlugin.git
```

(or the equivalent for your VCS)

and then immediately run `@load YourPlugin`.

#### With all your plugins in the same repository

By running `git init` in a new folder, then call *supybot-plugin-wizard* in this folder to create each plugin in a subdirectory. This is the easiest for your if you maintain many plugins, as you don't have to manage many repositories.

First, users should install a client for your VCS, and dependencies of your plugin(s) they want to use, if any.

Users can download all your plugins at once:

```
cd ~/
git clone https://example.org/~jdoe/LimnoriaPlugins.git JdoePlugins
```

and configure their bot to look for plugins in this directory:

```
@config supybot.directories.plugins [config supybot.directories.plugins], /home/me/
→JdoePlugins
```

Alternatively, to make it easier for your users, you can add your repository to Limnoria's list of known repository, by sending a pull request to: <https://github.com/ProgVal/Limnoria/blob/master/plugins/PluginDownloader/plugin.py>

---

**Note:** PluginDownloader currently only works on GitHub, but pull requests to add support for other forges are welcome.

---

Once it is accepted, users can then download your plugin in a single step instead of the commands above:

```
@pluginDownloader install Jdoe
```

Either way, they can now run `@load YourPlugin`

### 2.6.2 Via pip / PyPI

This requires a little more work for you, but uses mainstream Python package management (pip/PyPI) so it is a lot easier for users (as they don't have to use a VCS, put your plugin in the right directory, or install dependencies manually)

**Warning:** The following section assumes both your and your users have Limnoria 2020.05.08 or newer.

#### Setting up your plugin

To make your plugin installable via pip, you first need to create a `setup.py` file in the same directory as the other files of your plugin. You have probably seen one already if you are a Python developer. Limnoria provides a small wrapper over `setuptools.setup`, so you don't have to type in most of the fields. The minimal `setup.py` for a plugin named `YourPlugin` is:

```
from supybot.setup import plugin_setup

plugin_setup(
    'YourPlugin',
)
```

This will automatically populate:

- the package name (`name = "limnoria-yourplugin"`)
- author information (`author` and `author_email` if they are set in `__init__.py`)
- maintainer information (`idem`)
- version and URL (based on `__version__` and `__url__` in `__init__.py`)

- `package = "limnoria_yourplugin"`
- `package_dirs = {"limnoria_yourplugin": "."}`
- add Limnoria as dependency
- etc.

You can add any `setuptools` fields you like. For example, to add `requests` as a dependency:

```
from supybot.setup import plugin_setup

plugin_setup(
    'YourPlugin',
    install_requires=[
        'requests',
    ],
)
```

If you don't like the name *limnoria-yourplugin*, you can also override it. For example:

```
from supybot.setup import plugin_setup

plugin_setup(
    'YourPlugin',
    name='limnoria-this-is-my-plugin',
)
```

## Installing plugins

Finally, once you pushed your plugin, users can install it simply with:

```
pip3 install git+https://example.org/~jdoe/YourPlugin.git
```

Or, if you use a single repository for multiple plugins:

```
pip3 install "git+https://example.org/~jdoe/Supybot-plugins.git
↳#subdirectory=YourPlugin"
```

and this will automatically install your plugin's dependencies as well. Then, they just need to run `@load YourPlugin` as usual.

For example, to install the LinkRelay plugin from <https://github.com/progval/Supybot-plugins>:

```
pip3 install "git+https://github.com/progval/Supybot-plugins.git
↳#subdirectory=LinkRelay"
```

## Publishing your plugin (optional)

Additionally, you may want to publish your plugin to PyPI, to make it easier for users to install.

First, you must create an account on <https://pypi.org/> and install `twine`:

```
python3 -m pip install --user --upgrade twine
```

Then, you can generate and publish your plugin:

```
python3 -m twine sdist
python3 -m twine upload dist/*
```

And every time you want to publish an upgrade, update the version in `__init__.py` and run this last command again.

For more details, see the official Python documentation on:

- [generating archives](#)
- [uploading archives](#)

And users can simply install it with:

```
sudo pip3 install limnoria-yourplugin
```

## 2.7 Using Supybot's utils module

Supybot provides a wealth of utilities for plugin writers in the `supybot.utils` module, this tutorial describes these utilities and shows you how to use them.

### 2.7.1 str.py

#### The Format Function

The `supybot.utils.str` module provides a bunch of utility functions for handling string values. This section contains a quick rundown of all of the functions available, along with descriptions of the arguments they take. First and foremost is the `format` function, which provides a lot of capability in just one function that uses string-formatting style to accomplish a lot. So much so that it gets its own section in this tutorial. All other functions will be in other sections. `format` takes several arguments - first, the format string (using the format characters described below), and then after that, each individual item to be formatted. Do not attempt to use the `%` operator to do the formatting because that will fall back on the normal string formatting operator. The `format` function uses the following string formatting characters.

- `%` - literal `%`
- `i` - integer
- `s` - string
- `f` - float
- `r` - repr
- `b` - form of the verb `to be` (takes an int)
- `h` - form of the verb `to have` (takes an int)
- `L` - commaAndify (takes a list of strings or a tuple of ([strings], and))
- `p` - pluralize (takes a string)
- `q` - quoted (takes a string)
- `n` - n items (takes a 2-tuple of (n, item) or a 3-tuple of (n, between, item))
- `S` - a human-readable size (takes an int)
- `t` - time, formatted (takes an int)
- `T` - time delta, formatted (takes an int)

- u - url, wrapped in braces
- v - void, takes one or many arguments, but doesn't display it (useful for translation)

Here are a few examples to help elaborate on the above descriptions:

```
>>> format("Error %q has been reported %n. For more information, see %u.",
           "AttributeError", (5, "time"), "https://limnoria.net")

'Error "AttributeError" has been reported 5 times. For more information,
see <https://limnoria.net>.'
```

```
>>> i = 4
>>> format("There %b %n at this time. You are only allowed %n at any given
           time", i, (i, "active", "thread"), (5, "active", "thread"))
'There are 4 active threads at this time. You are only allowed 5 active
threads at any given time'
```

```
>>> i = 1
>>> format("There %b %n at this time. You are only allowed %n at any given
           time", i, (i, "active", "thread"), (5, "active", "thread"))
'There is 1 active thread at this time. You are only allowed 5 active
threads at any given time'
```

```
>>> ops = ["foo", "bar", "baz"]
>>> format("The following %n %h the %s capability: %L", (len(ops), "user"),
           len(ops), "op", ops)
'The following 3 users have the op capability: foo, bar, and baz'
```

As you can see, you can combine all sorts of combinations of formatting strings into one. In fact, that was the major motivation behind format. We have specific functions that you can use individually for each of those formatting types, but it became much easier just to use special formatting chars and the format function than concatenating a bunch of strings that were the result of other `utils.str` functions.

## The Other Functions

These are the functions that can't be handled by format. They are sorted in what I perceive to be the general order of usefulness (and I'm leaving the ones covered by format for the next section).

- `ellipsisify(s, n)` - Returns a shortened version of a string. Produces up to the first n chars at the nearest word boundary.
  - s: the string to be shortened
  - n: the number of characters to shorten it to
- `perlReToPythonRe(s)` - Converts a Perl-style regexp (e.g., `"/abcd/i"` or `"m/abcd/i"`) to an actual Python regexp (an `re` object)
  - s: the regexp string
- `perlReToReplacer(s)` - converts a perl-style replacement regexp (eg, `"s/foo/bar/g"`) to a Python function that performs such a replacement
  - s: the regexp string
- `dqrepr(s)` - Returns a `repr()` of s guaranteed to be in double quotes. (Double Quote Repr)
  - s: the string to be double-quote `repr()`'ed

- `toBool(s)` - Determines whether or not a string means True or False and returns the appropriate boolean value. True is any of “true”, “on”, “enable”, “enabled”, or “1”. False is any of “false”, “off”, “disable”, “disabled”, or “0”.
  - `s`: the string to determine the boolean value for
- `rsplit(s, sep=None, maxsplit=-1)` - functionally the same as `str.split` in the Python standard library except splitting from the right instead of the left. Python 2.4 has `str.rsplit` (which this function defers to for those versions  $\geq$  2.4), but Python 2.3 did not.
  - `s`: the string to be split
  - `sep`: the separator to split on, defaults to whitespace
  - `maxsplit`: the maximum number of splits to perform, -1 splits all possible splits.
- `normalizeWhitespace(s)` - reduces all multi-spaces in a string to a single space
  - `s`: the string to normalize
- `depluralize(s)` - the opposite of pluralize
  - `s`: the string to deppluralize
- `unCommaThe(s)` - Takes a string of the form “foo, the” and turns it into “the foo”
  - `s`: string, the
- `distance(s, t)` - computes the levenshtein distance (or “edit distance”) between two strings
  - `s`: the first string
  - `t`: the second string
- `soundex(s, length=4)` - computes the soundex for a given string
  - `s`: the string to compute the soundex for
  - `length`: the length of the soundex to generate
- `matchCase(s1, s2)` - Matches the case of the first string in the second string.
  - `s1`: the first string
  - `s2`: the string which will be made to match the case of the first

## The Commands Format Already Covers

These commands aren’t necessary because you can achieve them more easily by using the format command, but they exist if you decide you want to use them anyway though it is greatly discouraged for general use.

- `commaAndify(seq, comma=“,”, And=“and”)` - transforms a list of items into a comma separated list with an “and” preceding the last element. For example, [“foo”, “bar”, “baz”] becomes “foo, bar, and baz”. Is smart enough to convert two-element lists to just “item1 and item2” as well.
  - `seq`: the sequence of items (don’t have to be strings, but need to be ‘str()’-able)
  - `comma`: the character to use to separate the list
  - `And`: the word to use before the last element
- `pluralize(s)` - Returns the plural of a string. Put any exceptions to the general English rules of pluralization in the plurals dictionary in `supybot.utils.str`.
  - `s`: the string to pluralize

- `nItems(n, item, between=None)` - returns a string that describes a given number of an item (with any string between the actual number and the item itself), handles pluralization with the `pluralize` function above. Note that the arguments here are in a different order since `between` is optional.
  - `n`: the number of items
  - `item`: the type of item
  - `between`: the optional string that goes between the number and the type of item
- `quoted(s)` - Returns the string surrounded by double-quotes.
  - `s`: the string to quote
- `be(i)` - Returns the proper form of the verb “to be” based on the number provided (`be(1)` is “is”, `be(anything else)` is “are”)
  - `i`: the number of things that “be”
- `has(i)` - Returns the proper form of the verb “to have” based on the number provided (`has(1)` is “has”, `has(anything else)` is “have”)
  - `i`: the number of things that “has”

## 2.7.2 structures.py

### Intro

This module provides a number of useful data structures that aren’t found in the standard Python library. For the most part they were created as needed for the bot and plugins themselves, but they were created in such a way as to be of general use for anyone who needs a data structure that performs a like duty. As usual in this document, I’ll try and order these in order of usefulness, starting with the most useful.

### The queue classes

The `structures` module provides two general-purpose queue classes for you to use. The “`queue`” class is a robust full-featured queue that scales up to larger sized queues. The “`smallqueue`” class is for queues that will contain fewer (less than 1000 or so) items. Both offer the same common interface, which consists of:

- a constructor which will optionally accept a sequence to start the queue off with
- `enqueue(item)` - adds an item to the back of the queue
- `dequeue()` - removes (and returns) the item from the front of the queue
- `peek()` - returns the item from the front of the queue without removing it
- `reset()` - empties the queue entirely

In addition to these general-use queue classes, there are two other more specialized queue classes as well. The first is the “`TimeoutQueue`” which holds a queue of items until they reach a certain age and then they are removed from the queue. It features the following:

- `TimeoutQueue(timeout, queue=None)` - you must specify the timeout (in seconds) in the constructor. Note that you can also optionally pass it a queue which uses any implementation you wish to use whether it be one of the above (`queue` or `smallqueue`) or if it’s some custom queue you create that implements the same interface. If you don’t pass it a queue instance to use, it will build its own using `smallqueue`.
  - `reset()`, `enqueue(item)`, `dequeue()` - all same as above queue classes
  - `setTimeout(secs)` - allows you to change the timeout value

And for the final queue class, there's the "MaxLengthQueue" class. As you may have guessed, it's a queue that is capped at a certain specified length. It features the following:

- `MaxLengthQueue(length, seq=())` - the constructor naturally requires that you set the max length and it allows you to optionally pass in a sequence to be used as the starting queue. The underlying implementation is actually the queue from before.
  - `enqueue(item)` - adds an item onto the back of the queue and if it would push it over the max length, it dequeues the item on the front (it does not return this item to you)
  - all the standard methods from the queue class are inherited for this class

## The Other Structures

The most useful of the other structures is actually very similar to the "MaxLengthQueue". It's the "RingBuffer", which is essentially a `MaxLengthQueue` which fills up to its maximum size and then circularly replaces the old contents as new entries are added instead of dequeuing. It features the following:

- `RingBuffer(size, seq=())` - as with the `MaxLengthQueue` you specify the size of the `RingBuffer` and optionally give it a sequence.
  - `append(item)` - adds item to the end of the buffer, pushing out an item from the front if necessary
  - `reset()` - empties out the buffer entirely
  - `resize(i)` - shrinks/expands the `RingBuffer` to the size provided
  - `extend(seq)` - append the items from the provided sequence onto the end of the `RingBuffer`

The next data structure is the `TwoWayDictionary`, which as the name implies is a dictionary in which key-value pairs have mappings going both directions. It features the following:

- `TwoWayDictionary(seq=(), **kwargs)` - Takes an optional sequence of (key, value) pairs as well as any key=value pairs specified in the constructor as initial values for the two-way dict.
  - other than that, no extra features that a normal Python dict doesn't already offer with the exception that any (key, val) pair added to the dict is also added as (val, key) as well, so the mapping goes both ways. Elements are still accessed the same way you always do with Python 'dict's.

There is also a `MultiSet` class available, but it's very unlikely that it will serve your purpose, so I won't go into it here. The curious coder can go check the source and see what it's all about if they wish (it's only used once in our code, in the Relay plugin).

### 2.7.3 web.py

The web portion of Supybot's `utils` module is mainly used for retrieving data from websites but it also has some utility functions pertaining to HTML and email text as well. The functions in `web` are listed below, once again in order of usefulness.

- `getUrl(url, size=None, headers=None)` - gets the data at the URL provided and returns it as one large string
  - `url`: the location of the data to be retrieved or a `urllib2.Request` object to be used in the retrieval
  - `size`: the maximum number of bytes to retrieve, defaults to `None`, meaning that it is to try to retrieve all data
  - `headers`: a dictionary mapping header types to header data
- `getUrlFd(url, headers=None)` - returns a file-like object for a url
  - `url`: the location of the data to be retrieved or a `urllib2.Request` object to be used in the retrieval



- headers: a dictionary mapping header types to header data
- `htmlToText(s, tagReplace=" ")` - strips out all tags in a string of HTML, replacing them with the specified character
  - s: the HTML text to strip the tags out of
  - tagReplace: the string to replace tags with
- `strError(e)` - pretty-printer for web exceptions, returns a descriptive string given a web-related exception
  - e: the exception to pretty-print
- `mungeEmail(s)` - a naive e-mail obfuscation function, replaces “@” with “AT” and “.” with “DOT”
  - s: the e-mail address to obfuscate
- `getDomain(url)` - returns the domain of a URL - url: the URL in question

## 2.7.4 The Best of the Rest

### Intro

Rather than document each of the remaining portions of the `supybot.utils` module, I’ve elected to just pick out the choice bits from specific parts and document those instead. Here they are, broken out by module name.

### `supybot.utils.file` - file utilities

- `touch(filename)` - updates the access time of a file by opening it for writing and immediately closing it
- `mktemp(suffix="")` - creates a decent random string, suitable for a temporary filename with the given suffix, if provided
- the `AtomicFile` class - used for files that need to be atomically written, i.e., if there’s a failure the original file remains unmodified. For more info consult `file.py` in `src/utils`

### `supybot.utils.gen` - general utilities

- `timeElapsed(elapsed, [lots of optional args])` - given the number of seconds elapsed, returns a string with the English description of the amount of time passed, consult `gen.py` in `src/utils` for the exact argument list and documentation if you feel you could use this function.
- `exnToString(e)` - improved exception-to-string function. Provides nicer output than a simple `str(e)`.
- `InsensitivePreservingDict` class - a dict class that is case-insensitive when accessing keys

### `supybot.utils.iter` - iterable utilities

- `len(iterable)` - returns the length of a given iterable
- `groupby(key, iterable)` - equivalent to the `itertools.groupby` function available as of Python 2.4. Provided for backwards compatibility.
- `any(p, iterable)` - Returns true if any element in the iterable satisfies the predicate p
- `all(p, iterable)` - Returns true if all elements in the iterable satisfy the predicate p
- `choice(iterable)` - Returns a random element from the iterable

### supybot.dynamicScope / dynamic - accessing variables in the stack

This feature is not in *supybot.utils* but still deserves to be documented as a utility.

Although you should avoid using this feature as long as you can, it is sometimes necessary to access variables the Supybot API does not provide you.

For instance, the *Aka* plugin provides per-channel aliases by overriding *getCommandMethod*. However, the channel where the command is called is not passed to this functions, so when writing *Aka* I could either add this parameter (and thus break all plugins all plugins already overriding this method) or use this hack. I choosed this hack.

How does it work? This is quite simple: `dynamic.channel` is a shortcut for `supybot.dynamicScope.DynamicScope.__getattr__('channel')`, which browse the call stack backwards, looking for a variable named `channel`, and then returns is as far as it finds it (and returns `None` if there is no such variale).

Note that you don't have to import `dynamicScope`, the `dynamic` object is automatically set as a global variable when Supybot starts.

## 2.8 Capabilities

---

**Note:** I wrote this section with the little knowledge I have of the capabilities system; I work mainly by testing different possibilities until I get what I want. As for all the documentation, feel free to contact me to correct/enhance it.

---

First, you should know how capabilities work *on the user side*.

### 2.8.1 Checking for a capability given its name

You only have to use `ircdb.checkCapability(prefix, 'capability')`.

You can also override some behavior of the capability system. Here is the complete documentation of `ircdb.checkCapabiltiy`:

`supybot.ircdb.checkCapability` (*hostmask*, *capability*, *users*=<*supybot.ircdb.UsersDictionary object*>, *channels*=<*supybot.ircdb.ChannelsDictionary object*>, *ignoreOwner*=*False*, *ignoreChannelOp*=*False*, *ignoreDefaultAllow*=*False*)

Checks that the user specified by name/hostmask has the capability given.

*users* and *channels* default to `ircdb.users` and `ircdb.channels`.

*ignoreOwner*, *ignoreChannelOp*, and *ignoreDefaultAllow* are used to override default behavior of the capability system in special cases (actually, in the *AutoMode* plugin):

- *ignoreOwner* disables the behavior “owners have all capabilites”
- *ignoreChannelOp* disables the behavior “channel ops have all channel capabilities”
- *ignoreDefaultAllow* disables the behavior “if a user does not have a capability or the associated anticapability, then they have the capability”

### 2.8.2 Manipulating capability names

Although you can manipulate capability names with string operations, Supybot provides a few methods to do that “in the abstract” (could be useful if we change the capability syntax one day...):

```

supybot.ircdb.isCapability (capability)
supybot.ircdb.makeChannelCapability (channel, capability)
    Makes a channel capability given a channel and a capability.
supybot.ircdb.isChannelCapability (capability)
    Returns True if capability is a channel capability; False otherwise.
supybot.ircdb.makeAntiCapability (capability)
    Returns the anticapability of a given capability.
supybot.ircdb.unAntiCapability (capability)
    Takes an anticapability and returns the non-anti form.
supybot.ircdb.invertCapability (capability)
    Make a capability into an anticapability and vice versa.
supybot.ircdb.isAntiCapability (capability)
    Returns True if capability is an anticapability; False otherwise.
supybot.ircdb.canonicalCapability (capability)

```

## 2.9 Special methods and catching events

This page is a non-exhaustive list of special plugin method names and events catchable via those methods (other events include *configuration hooks* and *HTTP server callbacks*)

All methods here are defined in `supybot-callbacks-plugin`. You may override them if you need, but make sure you call the parent's one unless you actually don't want to do it.

In case multiple plugins implement the same special methods, the order they are called depends on the `callAfter` and `callBefore` (lists of plugin names) attributes of the plugins, if they are set.

### 2.9.1 Loading and unloading

The `__init__` method gets called with an `Irc` object as a parameter when a plugin is loaded (or has just been reloaded). Make sure you always call the parent's `__init__`.

When a plugin is unloaded (or is to be reloaded), the `die` method is called (with no parameter). Also make sure you always call the parent's `die`.

### 2.9.2 Commands and numerics

You can catch commands directly with “do-methods”: when the bot receives a `PRIVMSG`, all `doPrivmsg` methods are called; when it gets a 437 message, all `do437` methods are called, etc.

Those command take two commands: an *Irc object* and a *IrcMsg object*.

To get a list of all possible messages, check [IRC specifications](#).

### 2.9.3 Filters

The `inFilter` and `outFilter` methods allow you to “intercept” messages between the bot and the network and to alter them.

`inFilter` gets messages just after they are parsed from network; and its return value is fed to the bot. `outFilter` does the opposite: it get any message the bot is about to send, and returns a message (which may be the same) that will be sent instead.

### 2.9.4 Commands handling

#### Command dispatching

---

**Note:** I wrote this subsection with the little knowledge I have of the commands handling (all I know comes from hacks I made to write the Aka plugin), so keep in mind some informations might be wrong. As for all the documentation, feel free to contact me to correct/enhance it.

---

- `isCommandMethod` takes a command name as a string (which may contain spaces) and returns a boolean telling if the plugin provides this command.
- `listCommands` returns a list of command names as strings (which may contain spaces)
- `getCommand` takes a potential command name as a list of strings, and returns a truncated list corresponding to the name of a command provided by the plugin. If no command match, it returns an empty list.
- `getCommandMethod` takes a command name as a list of strings and returns the corresponding method/function.
- `callCommand` gets a command name as a list of strings, an `irc` object, an `msg` object, and extra arguments (with `*args` and `**kwargs`), calls `getCommandMethod` to get the command method, and calls it with the arguments. It also calls the functions in `pre_command_callback`.

#### Pre-command-call callbacks

---

**Note:** Until stock Supybot and Gribble merge this feature, this section only applies to Limnoria

---

If you want a function of your plugin to be called before every command call, you can add it to the `pre_command_callback` attribute of your plugin (actually, it is a static class attribute, so make sure you *add* it to the list and don't touch other items of the list).

At every command call, *all* callbacks are called, and if *any* of them returns `True`, the command is not called.

#### Other command-related events

- all `invalidCommand` methods get called (with an `Irc` object, an `IrcMsg` object, and a list of token) when a user calls a command that no plugin provides.

### 2.9.5 Regular expression triggered events

The `supybot.callbacks.PluginRegexp` class provides some utilities for creating plugins that act on regular expression matching.

## 2.10 Using Limnoria's HTTP server in your plugins

### 2.10.1 Introduction

---

**Note:** This feature is specific to Limnoria and not available in stock Supybot or Gribble.

---

Limnoria provides an HTTP server to plugins. This is not relevant for most plugins, but some of them have to start a server (either for serving a website or for being remotely called). The HTTP server provided by Limnoria aims at starting a single server for all of them, which means less used port and less resources usage.

Some example plugins are [Factoids](#), [WebStats](#), [GitHub](#), [UfrPaste](#), and [WebDoc](#)

### 2.10.2 Using the HTTP server in a plugin

Let's try to make a basic dictionary about Supybot! We'll call it Supystory.

We want to get plain text information about Supybot, Gribble, and Limnoria when accessing <http://localhost:8080/supystory/supybot>, <http://localhost:8080/supystory/gribble>, and <http://localhost:8080/supystory/limnoria>, an index page, and an HTML error page if the page is not found

#### Importing the HTTP server

One only have to add this line:

```
from supybot import httpserver
```

#### Creating a callback

If you are familiar with *BaseHTTPServer*, you will recognize the design, except you don't need to subclass *BaseHTTPServer*, because I already did it in *supybot.httpserver*.

Now, you have to subclass *httpserver.SupyHTTPServerCallback*. A callback is pretty much like an handler, but this is not an handler, because a callback is called by the handler.

Here is how to do it:

```
class SupystoryServerCallback(httpserver.SupyHTTPServerCallback):
    name = 'Supystory'
```

Now, you have to register the callback, because the HTTP server does not know what subdirectory it should assign to your callback. Do it with adding a `__init__` to your plugin (read Supybot's docs about it, for more informations):

```
class Supystory(callbacks.Plugin):
    def __init__(self, irc):
        # Some stuff needed by Supybot
        super().__init__(irc)

        # registering the callback
        callback = SupystoryServerCallback() # create an instance of the callback
        httpserver.hook('supystory', callback) # register the callback at '/supystory'
```

By the way, don't forget to unhook your callback when unloading your plugin, unless what it will be impossible to reload the plugin! Append this code to the following:

```
def die(self):
    # unregister the callback
    httpserver.unhook('supystory') # unregister the callback hooked at /supystory

    # Stuff for Supybot
    super().die()
```

Now, you can load your plugin, and you'll see on the server a beautiful link to `/supystory` called *Supystory*.

## Overriding the default error message

But our plugin does not do anything for the moment. If click the link, you'll get this message:

This **is** a default response of the Supybot HTTP server. If you see this message, it probably means you are developping a plugin, **and** you have neither overridden this message **or** defined an handler **for** this query.

That mean your browser sent a GET request, but you didn't teach your plugin how to handle it. First, we'll change this error message. Here is a new code for your callback:

```
class SupystoryServerCallback(httpserver.SupyHTTPServerCallback):
    name = 'Supystory'
    defaultResponse = ""
    This plugin handles only GET request, please don't use other requests."
```

Now, you'll get your customized message. But your plugin still doesn't work. You need to implement the GET request.

## Implementing the GET request

As I said before, callbacks are pretty much like handlers. In order to handle GET requests, you have to implement a method called... *doGet* (if you used `BaseHTTPServer`, you will notice this is not `do_GET`, because *doGet* is more homogeneous with Supybot naming style: *doPrivmsg*, *doPing*, and so on).

You will get the handler and the URI as arguments. The handler is a `BaseHTTPRequestHandler`, and the URI is a string.

Here is the code of the callback... pretty much simple, as ever:

```
class SupystoryServerCallback(httpserver.SupyHTTPServerCallback):
    name = 'Supystory'
    defaultResponse = ""
    This plugin handles only GET request, please don't use other requests."

    def doGet(self, handler, path):
        if path == '/supybot':
            response = b'Supybot is the best IRC bot ever.'
        elif path == '/gribble':
            response = b'Thanks to Gribble, we have many bug fixes and SQLite 3_
↪support'
        elif path == '/limnoria':
            response = b'Thanks to Limnoria, you can to internationalize your_
↪plugins and write a web server.'
        elif path == '' or path == '/':
```

(continues on next page)

(continued from previous page)

```

        handler.send_response(200) # Found
        handler.send_header('Content-type', 'text/html') # This is the MIME for_
↪HTML data
        handler.end_headers() # We won't send more headers
        handler.wfile.write(b"""
        <!DOCTYPE html>
        <html>
        <head>
        <meta charset="UTF-8">
        <title>Supystory</title>
        </head>
        <body>
        <h1>Supystory</h1>
        <p>
        Here are some links you can visit:
        <a href="/supybot">Supybot</a>
        <a href="/gribble">Gribble</a>
        <a href="/limnoria">Limnoria</a>
        </p>
        </body>
        </html>""")
        return
    else:
        handler.send_response(404) # Not found
        handler.send_header('Content-type', 'text/html') # This is the MIME for_
↪HTML data
        handler.end_headers() # We won't send more headers
        handler.wfile.write(b"""
        <!DOCTYPE html>
        <html>
        <head>
        <meta charset="UTF-8">
        <title>Error</title>
        </head>
        <body>
        <h1>404 Not found</h1>
        <p>
        The document could not be found. Try one of this links:
        <a href="/supybot">Supybot</a>
        <a href="/gribble">Gribble</a>
        <a href="/limnoria">Limnoria</a>
        </p>
        </body>
        </html>""")
        return
    handler.send_response(200)
    handler.send_header('Content-type', 'text/plain') # This is the MIME for_
↪plain text
    handler.end_headers() # We won't send more headers
    handler.wfile.write(response)

```

## Using templates

You may also want to allow your plugin's users to customize the web pages without editing the source code of the plugin itself.

Limnoria provides a template facility, which takes a file name, returns the content of a file from the file system if it exists (the user-defined template), and a default one otherwise (the developer's default template). does not exist.

In our case, we will do it only for the home page and the error page (which are the only 'big' pages), like this:

```

DEFAULT_TEMPLATES = {
    'supystory/index.html': """
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Supystory</title>
    </head>
    <body>
        <h1>Supystory</h1>
        <p>
            Here are some links you can visit:
            <a href="/supybot">Supybot</a>
            <a href="/gribble">Gribble</a>
            <a href="/limnoria">Limnoria</a>
        </p>
    </body>
</html>""",
    'supystory/error.html': """
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Error</title>
    </head>
    <body>
        <h1>404 Not found</h1>
        <p>
            The document could not be found. Try one of this links:
            <a href="/supybot">Supybot</a>
            <a href="/gribble">Gribble</a>
            <a href="/limnoria">Limnoria</a>
        </p>
    </body>
</html>""",
}

httpserver.set_default_templates(DEFAULT_TEMPLATES)

class SupystoryServerCallback(httpserver.SupyHTTPServerCallback):
    name = 'Supystory'
    defaultResponse = """
    This plugin handles only GET request, please don't use other requests."""

    def doGet(self, handler, path):
        if path == '/supybot':
            response = b'Supybot is the best IRC bot ever.'
        elif path == '/gribble':
            response = b'Thanks to Gribble, we have many bug fixes and SQLite 3_
↪support'
        elif path == '/limnoria':

```

(continues on next page)



(continued from previous page)

```

        response = b'Thanks to Limnoria, you can to internationalize your
↳plugins and write a web server.'
        elif path == '' or path == '/':
            handler.send_response(200) # Found
            handler.send_header('Content-type', 'text/html') # This is the MIME for
↳HTML data
            handler.end_headers() # We won't send more headers
            handler.wfile.write(httpserver.get_template('supystory/index.html').
↳encode('utf8'))
            return
        else:
            handler.send_response(404) # Not found
            handler.send_header('Content-type', 'text/html') # This is the MIME for
↳HTML data
            handler.end_headers() # We won't send more headers
            handler.wfile.write(httpserver.get_template('supystory/error.html').
↳encode('utf8'))
            return
            handler.send_response(200)
            handler.send_header('Content-type', 'text/plain') # This is the MIME for
↳plain text
            handler.end_headers() # We won't send more headers
            handler.wfile.write(response)

```

Then, the user can change the template by copying `data/web/supystory/index.html.example` to `data/web/supystory/index.html` and editing it. (Same for `error.html`.)

## 2.11 Event scheduling using supybot.schedule

```

###
# This is an example plugin that sends a message to a channel every 60 seconds,
# includes commands to stop, start, and reset the spammer, and a command to
# schedule a one-off event
###

# these are the default plugin modules
import supybot.utils as utils
from supybot.commands import *
import supybot.plugins as plugins
import supybot.ircutils as ircutils
import supybot.callbacks as callbacks
# these are the extra modules we'll be using
import time
import supybot.ircmsgs as ircmsgs
import supybot.schedule as schedule

class Spam(callbacks.Plugin):
    """Add the help for "@plugin help Spam" here
    This should describe *how* to use this plugin."""

    def __init__(self, irc):
        super().__init__(irc)
        # this is the channel we want to spam, and how frequently we want to do it.
        # It would be nicer to put it in a supybot config variable instead, but for

```

(continues on next page)

(continued from previous page)

```

# this demonstration, defining it in the plugin itself is fine.
self.spamChannel = '#testytest'
self.spamTime = 60
# scheduler events are global, so we want to test to make sure the event doesn
→ 't
# already exist. That is, even if the plugin is reloaded, the event sticks
# around. That means that you also have to be a little careful with your
# event names, especially if you have multiple plugins adding events. It also
# means that events will stick around even if the plugin they originated in
# is unloaded. I don't know how to delete them automatically on an unload,
→ but
# it's not normally an issue. Just make sure to stop the event before
→ unloading
# the plugin if that's what you want.
try:
    schedule.removeEvent('mySpamEvent')
except KeyError:
    pass
# now that we know there's no event by that name scheduled, we can create one.
# but first, we need to define a local helper function that will do the thing
# that we want. You can put the full contents into here, but I prefer to use
# separate methods, as it makes the code easier to get around in. We need
# the helper function because when you add events, you can't include
→ arguments.
def myEventCaller():
    self.spamEvent(irc)
# and now we can schedule the actual event
# schedule.addPeriodicEvent(f, t, name=None, now=True)
# f is the method, t is the time in seconds, name gives it a name and is
→ optional
# (but highly recommended, so that you can refer to the event in the future.
# otherwise, it's easy to accumulate duplicate events), and 'now' specifies
# whether to perform the action immediately, or to wait until time is up to
# perform it for the first time. Default is True.
schedule.addPeriodicEvent(myEventCaller, self.spamTime, 'mySpamEvent')
self.irc = irc

# make sure to have a capital letter or underscore or something, as it's not a
→ method
# that we want turned into an IRC command
def spamEvent(self, irc):
    # we need to use queueMsg() rather than reply(), because when the event is
    # scheduled on loading the plugin (as opposed to scheduling it with one of the
    # commands that we'll define next), it receives its irc object from __init__
→ ().
    # When the bot is started, the irc object that comes from __init__() doesn't
    # include a reply() method, because it's not loading in response to a command;
    # it's loading on the bot startup. If you don't want your event to be
→ scheduled
    # automatically and so don't schedule it from __init__(), but only from an IRC
    # command, then it's safe to use irc.reply(), as there are no circumstances
    # under which the irc object won't have a reply() method.
    irc.queueMsg(ircmsgs.privmsg(self.spamChannel, 'I\'m spamming the channel!'))

def start(self, irc, msg, args):
    """takes no arguments

```

(continues on next page)

(continued from previous page)

```

    A command to start the spammer."""
    # don't forget to redefine the event wrapper
    def myEventCaller():
        self.spamEvent(irc)
    try:
        schedule.addPeriodicEvent(myEventCaller, self.spamTime, 'mySpamEvent',
↪False)
    except AssertionError:
        irc.reply('Error: the spammer was already running!')
    else:
        irc.reply('Spammer started!')
    start = wrap(start)

    def stop(self, irc, msg, args):
        """takes no arguments

        A command to stop the spammer."""
        try:
            schedule.removeEvent('mySpamEvent')
        except KeyError:
            irc.reply('Error: the spammer wasn\'t running!')
        else:
            irc.reply('Spammer stopped.')
    stop = wrap(stop)

    def reset(self, irc, msg, args):
        """takes no arguments

        Resets the spammer. Can be useful if something changes and you want the
        spam to reflect that. For example, if you defined the spamChannel as a
        supybot config, and changed it while the spammer was running, it would still
        keep going on the same channel until you reset it."""
        def myEventCaller():
            self.spamEvent(irc)
        try:
            schedule.removeEvent('mySpamEvent')
        except KeyError:
            irc.reply('Spammer wasn\'t running')
        schedule.addPeriodicEvent(myEventCaller, self.spamTime, 'mySpamEvent', False)
        irc.reply('Spammer reset sucessfully!')
    reset = wrap(reset)

    # Here's an example of a one-off event, scheduled by an IRC command
    def sayhi(self, irc, msg, args, delay):
        """&lt;time delay&gt;

        Says hi after the specified delay"""
        def myEventCaller():
            self.Hello(irc)
        # for a one-off event, the time is an absolute time, not relative. So we need
        # to get the current time and add to it however long we want to wait
        t = time.time() + delay
        # since we don't specify a name, we won't be able to reference the events in
        # the future, but that's ok, because these are one-off events, so even if you
        # do call it multiple times, it'll just reply that same number of times and
        # then stop. But in some circumstances you might want to name them. Just
        # remember that it'll give an AssertionError if you try to create two events

```

(continues on next page)

(continued from previous page)

```
# with the same name
schedule.addEvent(myEventCaller, t)
irc.reply('"hi" scheduled for %d seconds from now!' % delay)
sayhi = wrap(sayhi, ['positiveInt'])

def Hello(self, irc):
    # since the irc object is coming from an IRC command, rather than from __init__
    ↪_(),
    # it's guaranteed to have a reply() method, so it's safe to use that. It
    # might be better to to use queueMsg() instead, regardless, but I don't know
    # enough about the supybot internals to say whether one is preferred over
    # the other
    irc.reply('hi!')

Class = Spam
```

This example comes from the Gribble Wiki: <https://sourceforge.net/p/gribble/wiki/Supybot.schedule/history>

Copyright 2010, 2015, nanotube and quantumlemur licensed under the Creative Commons Attribution ShareAlike 3.0 Unported license and/or the GNU Free Documentation License v 1.3 or later

## 2.12 Software architecture

Limnoria abstracts its internal architecture away from plugins through its plugin API, which is enough most of the time. However, you may need want to understand its internal architecture, either to debug complex problems, provide advanced features that hook into the internals, contribute to the core, or simply out of curiosity.

This guide will try to walk you through this, assuming you are already familiar with using the bot and writing plugins (if not, see the *Capabilities* documentation, the *Writing Your First Limnoria Plugin*, and *Special methods and catching events*,

You should also be somewhat familiar with the [IRC protocol](#).

---

**Note:** This document is a work in progress and is still incomplete. As usual, feel free to ask any questions in #limnoria @ Libera.

---

### 2.12.1 Main loop and drivers

The main event loop is a very classic synchronous loop. It is defined in `scripts/supybot`, and essentially just this:

```
while world.ircs:
    try:
        drivers.run()
    except KeyboardInterrupt:
        # Handle Ctrl-C (trigger shutdown)
    except:
        # Handle other unhandled errors
```

Where `drivers.run()` does this:

```

for (name, driver) in _drivers.items():
    if name not in _deadDrivers:
        driver.run()
for name in _deadDrivers:
    # Remove the driver
while _newDrivers:
    (name, driver) = _newDrivers.pop()
    # add the new driver

```

Drivers are the sources of events in the main thread. In a normal Limnoria setup, there are two types of drivers: the Socket driver (which connects synchronously to IRC) and the *schedule driver* (which runs functions periodically, like cron). Historically, there was an alternative driver to connect to IRC, based on Twisted. It was deprecated, then removed in 2019, because Python's `socket` module became as powerful as Twisted as it gained support for `select()` and TLS.

Network drivers have a reference to a `irclib.Irc` object, and do three things in their `run()` method:

1. check the connection is still alive (and schedule a reconnect if not)
2. get new messages from their `supybot.irclib.Irc` instance (using `supybot.irclib.Irc.takeMsg()`) and send them to the network
3. get new messages from the network and pass them to their `supybot.irclib.Irc` (using `supybot.irclib.Irc.feedMsg()`)

The actual implementation of the current Socket driver is actually a little more complex than this, as all Socket driver instances cooperate to use `select()` together, but this is the rough idea. See `src/drivers/Socket.py` for details.

## 2.12.2 irclib

As we saw above, network drivers pass their messages to a class defined in `irclib`, which is where most of the IRC protocol implementation is.

Unlike most event-driven software (especially IRC implementation), Limnoria does not have hooks that are registered to call a function when a specific event/IRC command is received. Instead, event listeners receive all events, and inherit on `supybot.irclib.IrcCommandDispatcher`, which calls a specific method based on the IRC command name. For example, it calls the `doTopic` method when receiving a `TOPIC` message.

This dispatching is used both in the main IRC handling (`supybot.irclib.Irc`) and plugins (via `supybot.callbacks.PluginMixin`, which inherits `supybot.irclib.IrcCommandDispatcher`).

We saw above that the `supybot.irclib.Irc` object receives messages directly from the driver. It's also in charge of keeping track of other callbacks (ie. plugins) via `supybot.irclib.Irc.addCallback()` and passing every message to their `__call__` method (which then does the dispatching on its own again, as it inherits `supybot.irclib.IrcCommandDispatcher`).

As there are few callbacks (under a hundred plugins), this simple architecture is efficient enough.

Additionally, when receiving a message and before sending one, it iterates through the list of plugins and calls their `inFilter` and `outFilter` methods (respectively), if any.

If you look at the code of `supybot.irclib.Irc` and `supybot.irclib.IrcState`, you see they are mostly made of `doXxx` methods, which exhaustively implement every known IRC command, update some state, and optionally react to it by queuing messages.

### 2.12.3 Commands

Next is the callbacks system, mostly implemented in `supybot.callbacks`. This is where all the magic happens to make plugins so easy to write; it's also the most complex part of Limnoria and the hardest to understand, because everything is tightly interleaved.

TODO

### 2.12.4 Registry

TODO

### 2.12.5 Auto-documentation

TODO

## 2.13 Frequently Asked Questions

This section tries to cover all questions you may have as a plugin developer. (If you are a user, check out the [User FAQ](#) instead.)

### 2.13.1 Where can I find the user who called a command?

The `msg` object passed to all event method as well as command methods is an `supybot.ircmsgs.IrcMsg` object, which stores the content of the message, the nick and hostname of its author, etc. Check the documentation of `supybot.ircmsgs.IrcMsg` to see all available attributes.:w

### 2.13.2 Where can I find the hostname from a user's nick?

The `irc` object passed to all event method as well as command methods is an `supybot.irclib.Irc` object, use `irc.state.nickToHostmask`

### 2.13.3 How do I get channel modes when writing a plugin?

I want to know who's an op in a certain channel, or who's voiced, or what the modes on the channel are. How do I do that?

Everything you need is kept in a `ChannelState` object in an `IrcState` object in the `Irc` object your plugin is given. To see the ops in a given channel, for instance, you would do this:

```
irc.state.channels['#channel'].ops
```

To see a dictionary mapping mode chars to values (if any), you would do this:

```
irc.state.channels['#channel'].modes
```

From there, things should be self-evident.

## 2.14 Library reference

### 2.14.1 supybot.callbacks

#### Plugin

**class** `supybot.callbacks.Plugin(*args, **kwargs)`  
 Bases: `supybot.callbacks.PluginMixin`, `supybot.callbacks.Commands`

**Proxy**  
 alias of `NestedCommandsIrcProxy`

**callCommand** (`*args, **kwargs`)  
 Given a command name, gets the method with `getCommandMethod()` and calls it.

**callPrecedence** (`*args, **kwargs`)  
 Returns a pair of (callbacks to call before me, callbacks to call after me)

**die** (`*args, **kwargs`)  
 Makes the callback die. Called when the parent Irc object dies.

**dispatchCommand** (`command, args=None`)  
 Given a string 'command', dispatches to doCommand.

**getCommand** (`args, stripOwnName=True`)  
 Among all the commands in this Commands object, recursively searches for the command whose name is the longest substring of `args`, and returns its name, splitted on spaces.

**getCommandHelp** (`command, simpleSyntax=None`)  
 Returns the help string of the given command, using `getCommandMethod()`.

**getCommandMethod** (`command`)  
 Gets the given command from this plugin, using `getCommand()`. Plugins only need to implement this if they have a dynamic set of commands.

**inFilter** (`*args, **kwargs`)  
 Used for filtering/modifying messages as they're entering.  
  
 ircmsgs.IrcMsg objects are immutable, so this method is expected to return another ircmsgs.IrcMsg object. Obviously the same IrcMsg can be returned.

**isCommand** (`*args, **kwargs`)  
 Convenience, backwards-compatibility, semi-deprecated.

**isCommandMethod** (`name`)  
 Returns whether a given method name is a command in this plugin. Plugins only need to implement this if they have a dynamic set of commands.

**isDisabled** (`command`)  
 Returns whether the given `command` is disabled.

**listCommands** (`pluginCommands=[]`)  
 List all the commands in this Commands object. Plugins only need to implement this if they have a dynamic set of commands.

**name** ()  
 Returns the name of this Commands object (usually the plugin name).

**outFilter** (`*args, **kwargs`)  
 Used for filtering/modifying messages as they're leaving.

As with `inFilter`, an `IrcMsg` is returned.

**postTransition** (\*args, \*\*kwargs)

Called when the state of the IRC connection changes.

*msg* is the message that triggered the transition, if any.

**registryValue** (name, channel=None, network=None, \*, value=True)

Returns the value of a configuration variable specified by *name*.

If the configuration variable has a channel- or network-specific variable (ie. if its value can change across channels or networks), *channel* and *network* allow getting the most specific value. If neither is given, returns the generic value.

If *value=False*, returns the variable itself (an instance of `supybot.registry.Value`) instead of its value.

**reset** (\*args, \*\*kwargs)

Resets the callback. Called when reconnecting to the server.

**setRegistryValue** (name, value, channel=None, network=None)

Sets a configuration variable. See `registryValue()`

## PluginRegexp

**class** `supybot.callbacks.PluginRegexp` (\*args, \*\*kwargs)

Bases: `supybot.callbacks.Plugin`

Same as `Plugin`, except allows the user to also include regexp-based callbacks. All regexp-based callbacks must be specified in the `set` (or `list`) attribute “`regexps`”, “`addressedRegexps`”, or “`unaddressedRegexps`” depending on whether they should always be triggered, triggered only when the bot is addressed, or triggered only when the bot isn’t addressed.

**addressedRegexps** = ()

‘`addressedRegexps`’ methods are called only when the message is addressed, and then, only with the payload (i.e., what is returned from the ‘`addressed`’ function).

**regexps** = ()

‘`regexps`’ methods are called whether the message is addressed or not.

**unaddressedRegexps** = ()

‘`unaddressedRegexps`’ methods are called only when the message is *not* addressed.

## tokenize

`supybot.callbacks.tokenize` (*s*, channel=None, network=None)

A utility function to create a `Tokenizer` and tokenize a string.

## Other classes

This module contains the basic callbacks for handling PRIVMSGs.

**exception** `supybot.callbacks.ArgumentError`

Bases: `supybot.callbacks.Error`

The bot replies with a help message when this is raised.

**class** `supybot.callbacks.BasePlugin` (\*args, \*\*kwargs)

Bases: `object`



```

class supybot.callbacks.CanonicalNameDict (dict=None, key=None)
    Bases: supybot.utils.gen.InsensitivePreservingDict

    key (s)
        Override this if you wish.

class supybot.callbacks.CanonicalNameSet (iterable=())
    Bases: supybot.utils.gen.NormalizingSet

    normalize (s)

class supybot.callbacks.CanonicalString (default, *args, **kwargs)
    Bases: supybot.registry.NormalizedString

    normalize (s)

class supybot.callbacks.CommandProcess (target=None, args=(), kwargs={})
    Bases: supybot.world.SupyProcess

    Just does some extra logging and error-recovery for commands that need to run in processes.

    run ()
        Method to be run in sub-process; can be overridden in sub-class

class supybot.callbacks.CommandThread (target=None, args=(), kwargs={})
    Bases: supybot.world.SupyThread

    Just does some extra logging and error-recovery for commands that need to run in threads.

    run ()
        Method representing the thread's activity.

        You may override this method in a subclass. The standard run() method invokes the callable object passed
        to the object's constructor as the target argument, if any, with sequential and keyword arguments taken
        from the args and kwargs arguments, respectively.

class supybot.callbacks.Commands (*args, **kwargs)
    Bases: supybot.callbacks.BasePlugin, supybot.utils.python.SynchronizedAndFirewalled

    callCommand (*args, **kwargs)
        Given a command name, gets the method with getCommandMethod() and calls it.

    canonicalName ()
        Same as name(), but normalized.

    commandArgs = ['self', 'irc', 'msg', 'args']

    getCommand (args, stripOwnName=True)
        Among all the commands in this Commands object, recursively searches for the command whose name is
        the longest substring of args, and returns its name, splitted on spaces.

    getCommandHelp (command, simpleSyntax=None)
        Returns the help string of the given command, using getCommandMethod().

    getCommandMethod (command)
        Gets the given command from this plugin, using getCommand(). Plugins only need to implement this
        if they have a dynamic set of commands.

    isCommand (*args, **kwargs)
        Convenience, backwards-compatibility, semi-deprecated.

```

**isCommandMethod** (*name*)

Returns whether a given method name is a command in this plugin. Plugins only need to implement this if they have a dynamic set of commands.

**isDisabled** (*command*)

Returns whether the given `command` is disabled.

**listCommands** (*pluginCommands=[]*)

List all the commands in this `Commands` object. Plugins only need to implement this if they have a dynamic set of commands.

**name** ()

Returns the name of this `Commands` object (usually the plugin name).

**pre\_command\_callbacks** = []

**class** `supybot.callbacks.Disabled` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.SpaceSeparatedListOf`

**List**

alias of `CanonicalNameSet`

**Value**

alias of `CanonicalString`

**sorted** = `True`

**class** `supybot.callbacks.DisabledCommands`

Bases: `object`

**add** (*command, plugin=None*)

**disabled** (*command, plugin=None*)

**remove** (*command, plugin=None*)

**exception** `supybot.callbacks.Error`

Bases: `Exception`

Generic class for errors in Privmsg callbacks.

`supybot.callbacks.IrcObjectProxy`

alias of `supybot.callbacks.NestedCommandsIrcProxy`

**class** `supybot.callbacks.MetaSynchronizedAndFirewalled`

Bases: `supybot.log.MetaFirewall, supybot.utils.python.MetaSynchronized`

**class** `supybot.callbacks.NestedCommandsIrcProxy` (*irc, msg, args, nested=0, replyIrc=None*)

Bases: `supybot.callbacks.ReplyIrcProxy`

A proxy object to allow proper nesting of commands (even threaded ones).

**error** (*s=", Raise=False, \*\*kwargs*)

Replies with an error.

#### Parameters

- **s** (*str*) – The error message
- **Raise** (*bool*) – If True, this will raise `Error` that will propagate so that the caller of this function immediately terminates
- **\*\*kwargs** – See `NestedCommandsIrcProxy.reply()`'s keyword arguments

**evalArgs** (*withClass=None*)

**finalEval** ()

**findCallbacksForArgs** (*args*)

Returns a two-tuple of (command, plugins) that has the command (a list of strings) and the plugins for which it was a command.

**noReply** (*msg=None*)

**replies** (*L, prefixer=None, joiner=None, onlyPrefixFirst=False, to=None, oneToOne=None, \*\*kwargs*)

**reply** (*s, noLengthCheck=False, prefixNick=None, action=None, private=None, notice=None, to=None, msg=None, sendImmediately=False, stripCtcp=True*)

Keyword arguments:

#### Parameters

- **noLengthCheck** (*bool*) – True if the length shouldn't be checked (used for 'more' handling)
- **prefixNick** (*bool*) – False if the nick shouldn't be prefixed to the reply.
- **action** (*bool*) – True if the reply should be an action.
- **private** (*bool*) – True if the reply should be in private.
- **notice** (*bool*) – True if the reply should be noticed when the bot is configured to do so.
- **to** (*str*) – The nick or channel the reply should go to. Defaults to msg.args[0] (or msg.nick if private)
- **sendImmediately** (*bool*) – True if the reply should use sendMsg() which bypasses conf.supybot.protocols.irc.throttleTime and gets sent before any queued messages

**class** supybot.callbacks.PluginMixin (*irc*)

Bases: *supybot.callbacks.BasePlugin, supybot.irclib.IrcCallback*

#### Proxy

alias of *NestedCommandsIrcProxy*

**alwaysCall** = ()

**canonicalName** ()

**classModule** = None

**getPluginHelp** ()

**noIgnore** = False

**public** = True

**registryValue** (*name, channel=None, network=None, \*, value=True*)

Returns the value of a configuration variable specified by *name*.

If the configuration variable has a channel- or network-specific variable (ie. if its value can change across channels or networks), *channel* and *network* allow getting the most specific value. If neither is given, returns the generic value.

If *value=False*, returns the variable itself (an instance of *supybot.registry.Value*) instead of its value.

**setRegistryValue** (*name, value, channel=None, network=None*)

Sets a configuration variable. See *registryValue* ()

**setUserValue** (*name, prefixOrName, value, ignoreNoUser=True, setValue=True*)

**threaded** = **False**

**userValue** (*name, prefixOrName, default=None*)

`supybot.callbacks.Privmsg`

alias of `supybot.callbacks.Plugin`

`supybot.callbacks.PrivmsgCommandAndRegexp`

alias of `supybot.callbacks.PluginRegexp`

**class** `supybot.callbacks.ReplyIrcProxy` (*irc, msg, replyIrc=None*)

Bases: `supybot.callbacks.RichReplyMethods`

This class is a thin wrapper around an `irclib.Irc` object that gives it the `reply()` and `error()` methods (as well as everything in `RichReplyMethods`, based on those two).

If `replyIrc` is given in addition to `irc`, commands will be run on `irc` but replies will be delivered to `replyIrc`. This is used by the Network plugin to run commands on other networks.

**error** (*s, msg=None, \*\*kwargs*)

**getRealIrc** ()

Returns the real `irclib.Irc` object underlying this proxy chain.

**queueMultilineBatches** (*msgs, target, targetNick, concat, allowedLength=0, sendImmediately=False*)

Queues the `msgs` passed as argument in batches using draft/multiline batches.

This errors if `experimentalExtensions` is disabled or draft/multiline was not negotiated.

**reply** (*s, msg=None, \*\*kwargs*)

Keyword arguments:

#### Parameters

- **noLengthCheck** (*bool*) – True if the length shouldn't be checked (used for 'more' handling)
- **prefixNick** (*bool*) – False if the nick shouldn't be prefixed to the reply.
- **action** (*bool*) – True if the reply should be an action.
- **private** (*bool*) – True if the reply should be in private.
- **notice** (*bool*) – True if the reply should be noticed when the bot is configured to do so.
- **to** (*str*) – The nick or channel the reply should go to. Defaults to `msg.args[0]` (or `msg.nick` if private)
- **sendImmediately** (*bool*) – True if the reply should use `sendMsg()` which bypasses `conf.supybot.protocols.irc.throttleTime` and gets sent before any queued messages

**class** `supybot.callbacks.RichReplyMethods`

Bases: `object`

This is a mixin so these replies need only be defined once. It operates under several assumptions, including the fact that 'self' is an `Irc` object of some sort and there is a `self.msg` that is an `IrcMsg`.

**errorInvalid** (*what, given=None, s="", repr=True, \*\*kwargs*)

**errorNoCapability** (*capability, s="", \*\*kwargs*)

**errorNoUser** (*s="", name='that user', \*\*kwargs*)

**errorNotRegistered** (*s="", \*\*kwargs*)

**errorPossibleBug** (*s*="", *\*\*kwargs*)

**errorRequiresPrivacy** (*s*="", *\*\*kwargs*)

**noReply** (*msg=None*)

**replies** (*L*, *prefixer=None*, *joiner=None*, *onlyPrefixFirst=False*, *oneToOne=None*, *\*\*kwargs*)

**replyError** (*s*="", *\*\*kwargs*)

**replySuccess** (*s*="", *\*\*kwargs*)

Replies with a success message, configurable with `supybot.replies.success` or the `Success` plugin.

#### Parameters

- **s** (*str*) – Text to append to the standard success message
- **\*\*kwargs** – See `NestedCommandsIrcProxy.reply()`'s keyword arguments

**exception** `supybot.callbacks.SilentError`

Bases: `supybot.callbacks.Error`

An error that we should not notify the user.

`supybot.callbacks.SimpleProxy`

alias of `supybot.callbacks.ReplyIrcProxy`

**class** `supybot.callbacks.Tokenizer` (*brackets=""*, *pipe=False*, *quotes=""*)

Bases: `object`

**separators** = `'\x00\r\n \t'`

`supybot.callbacks.addressed` (*irc*, *msg*, *\*\*kwargs*)

If *msg* is addressed to 'name', returns the portion after the address. Otherwise returns the empty string.

`supybot.callbacks.canonicalName` (*command*, *preserve\_spaces=False*)

Turn a command into its canonical form.

Currently, this makes everything lowercase and removes all dashes and underscores.

`supybot.callbacks.checkCommandCapability` (*msg*, *cb*, *commandName*)

`supybot.callbacks.error` (*\*args*, *\*\*kwargs*)

`supybot.callbacks.formatCommand` (*command*)

`supybot.callbacks.getHelp` (*method*, *name=None*, *doc=None*)

`supybot.callbacks.getSyntax` (*method*, *name=None*, *doc=None*)

`supybot.callbacks.reply` (*\*args*, *\*\*kwargs*)

## 2.14.2 supybot.commands

Includes wrappers for commands.

**class** `supybot.commands.context` (*spec*)

Bases: `object`

**class** `supybot.commands.any` (*spec*, *continueOnError=False*)

Bases: `supybot.commands.context`

**class** `supybot.commands.many` (*spec*, *continueOnError=False*)

Bases: `supybot.commands.any`

```
class supybot.commands.optional(spec, default=None)
```

Bases: [supybot.commands.additional](#)

```
class supybot.commands.additional(spec, default=None)
```

Bases: [supybot.commands.context](#)

```
class supybot.commands.rest(spec)
```

Bases: [supybot.commands.context](#)

```
class supybot.commands.getopts(getopts)
```

Bases: [supybot.commands.context](#)

The empty string indicates that no argument is taken; None indicates that there is no converter for the argument.

```
class supybot.commands.first(*specs, **kw)
```

Bases: [supybot.commands.context](#)

```
class supybot.commands.reverse(spec)
```

Bases: [supybot.commands.context](#)

```
class supybot.commands.commalist(spec)
```

Bases: [supybot.commands.context](#)

```
supybot.commands.getConverter(name)
```

```
supybot.commands.addConverter(name, wrapper)
```

```
supybot.commands.callConverter(name, irc, msg, args, state, *L)
```

```
supybot.commands.urlSnarfer(f)
```

Protects the snarfer from loops (with other bots) and whatnot.

```
supybot.commands.thread(f)
```

Makes sure a command spawns a thread when called.

```
supybot.commands.wrap(f, *args, **kwargs)
```

Useful wrapper for plugin commands.

Valid converters are: admin, anything, banmask, boolean, callerInGivenChannel, capability, channel, channelDb, channelOrGlobal, channels, checkCapability, checkCapabilityButIgnoreOwner, checkChannelCapability, color, commandName, email, expiry, filename, float, glob, halfop, haveHalfop, haveHalfop+, haveOp, haveOp+, haveVoice, haveVoice+, hostmask, httpIri, httpUrl, id, inChannel, index, int, ip, iri, isGranted, letter, literal, long, lowered, matches, networkIrc, nick, nickInChannel, nonInt, nonNegativeInt, now, onlyInChannel, op, otherUser, owner, plugin, positiveInt, private, public, regexpMatcher, regexpMatcherMany, regexpReplacer, seenNick, something, somethingWithoutSpaces, text, to, url, user, validChannel, voice.

### Parameters

- **f** – A command, taking (self, irc, msg, args, ...) as arguments
- **specList** – A list of converters and contexts

```
supybot.commands.process(f, *args, **kwargs)
```

Runs a function <f> in a subprocess.

Several extra keyword arguments can be supplied. <pn>, the pluginname, and <cn>, the command name, are strings used to create the process name, for identification purposes. <timeout>, if supplied, limits the length of execution of target function to <timeout> seconds. <heap\_size>, if supplied, limits the memory used by the target function.

```
supybot.commands.regexp_wrapper(s, reobj, timeout, plugin_name, fcn_name)
```

A convenient wrapper to stuff regexp search queries through a subprocess.

This is used because specially-crafted regexps can use exponential time and hang the bot.

```
class supybot.commands.Spec(types, allowExtra=False)
    Bases: object
```

### 2.14.3 supybot.ircmsgs

This module provides the basic `IrcMsg` object used throughout the bot to represent the actual messages. It also provides several helper functions to construct such messages in an easier way than the constructor for the `IrcMsg` object (which, as you'll read later, is quite... full-featured :))

```
class supybot.ircmsgs.IrcMsg(s="", command="", args=(), prefix="", server_tags=None,
                             msg=None, reply_env=None)
    Bases: object
```

Class to represent an IRC message.

As usual, ignore attributes that begin with an underscore. They simply don't exist. Instances of this class are *not* to be modified, since they are hashable. Public attributes of this class are `.prefix`, `.command`, `.args`, `.nick`, `.user`, and `.host`.

The constructor for this class is pretty intricate. It's designed to take any of three major (sets of) arguments.

Called with no keyword arguments, it takes a single string that is a raw IRC message (such as one taken straight from the network).

Called with keyword arguments, it *requires* a `command` parameter. `Args` is optional, but with most commands will be necessary. `Prefix` is obviously optional, since clients aren't allowed (well, technically, they are, but only in a completely useless way) to send prefixes to the server.

Since this class isn't to be modified, the constructor also accepts a `'msg'` keyword argument representing a message from which to take all the attributes not provided otherwise as keyword arguments. So, for instance, if a programmer wanted to take a PRIVMSG they'd gotten and simply redirect it to a different source, they could do this:

```
IrcMsg(prefix="", args=(newSource, otherMsg.args[1]), msg=otherMsg)
```

#### **command**

The IRC command of the message (eg. PRIVMSG, NOTICE, MODE, QUIT, ...). In case of "split" commands (eg. CAP LS), this is only the first part, and the other parts are in `args`.

#### **args**

Arguments of the IRC command (including subcommands). For example, for a PRIVMSG, `args = ('#channel', 'content of the message')`.

#### **channel**

The name of the channel this message was received on or will be sent to; or `None` if this is not a channel message (PRIVMSG to a nick, QUIT, etc.)

`msg.args[0]` was formerly used to get the channel, but it had several pitfalls (such as needing server-specific channel vs nick detection, and needing to strip statusmsg characters).

#### **prefix**

`nick!user@host` of the author of the message, or `None`.

#### **nick**

Nickname of the author of the message, or `None`.

#### **user**

Username/ident of the author of the message, or `None`.

#### **host**

Hostname of the author of the message, or `None`.

**time**

Float timestamp of the moment the message was sent by the server. If the server does not support *server-time*, this falls back to the value of *time.time()* when the message was received.

**server\_tags**

Dictionary of IRCv3 message tags. *None* values indicate the tag is present but has no value.

This includes client tags; the name is meant to disambiguate wrt the *tags* attribute, which are tags used internally by Supybot/Limnoria.

**reply\_env**

(Mutable) dictionary of internal key:value pairs, all of which must be strings.

Several plugins offer string templating, such as the ‘echo’ command in the Misc plugin; which replace *\$variable* with a value.

Adding values to this dictionary allows access to these values from these commands; this is especially useful when nesting commands.

**tags**

(Mutable) dictionary of internal key:value pairs on this message.

This is not to be confused with IRCv3 message tags; these are stored as *server\_tags* (including the client tags).

**tag** (*tag*, *value=True*)

Affect an internal key:value pair to this message.

This is not to be confused with IRCv3 message tags; these are stored as *server\_tags* (including the client tags).

**tagged** (*tag*)

Get the value affected to a tag, or None if it is not set..

**exception** supybot.ircmsgs.**MalformedIrcMsg**

Bases: ValueError

supybot.ircmsgs.**action** (*recipient*, *s*, *prefix=""*, *msg=None*)

Returns a PRIVMSG ACTION to recipient with the message s.

supybot.ircmsgs.**ban** (*channel*, *hostmask*, *exception=""*, *prefix=""*, *msg=None*)

Returns a MODE to ban nick on channel.

supybot.ircmsgs.**bans** (*channel*, *hostmasks*, *exceptions=()*, *prefix=""*, *msg=None*)

Returns a MODE to ban each of nicks on channel.

supybot.ircmsgs.**dehalfop** (*channel*, *nick*, *prefix=""*, *msg=None*)

Returns a MODE to dehalfop nick on channel.

supybot.ircmsgs.**dehalfops** (*channel*, *nicks*, *prefix=""*, *msg=None*)

Returns a MODE to dehalfop each of nicks on channel.

supybot.ircmsgs.**deop** (*channel*, *nick*, *prefix=""*, *msg=None*)

Returns a MODE to deop nick on channel.

supybot.ircmsgs.**deops** (*channel*, *nicks*, *prefix=""*, *msg=None*)

Returns a MODE to deop each of nicks on channel.

supybot.ircmsgs.**devoice** (*channel*, *nick*, *prefix=""*, *msg=None*)

Returns a MODE to devoice nick on channel.

supybot.ircmsgs.**devoices** (*channel*, *nicks*, *prefix=""*, *msg=None*)

Returns a MODE to devoice each of nicks on channel.



`supybot.ircmsgs.halfop(channel, nick, prefix="", msg=None)`  
Returns a MODE to halfop nick on channel.

`supybot.ircmsgs.halfops(channel, nicks, prefix="", msg=None)`  
Returns a MODE to halfop each of nicks on channel.

`supybot.ircmsgs.invite(nick, channel, prefix="", msg=None)`  
Returns an INVITE for nick.

`supybot.ircmsgs.isAction(msg)`  
A predicate returning true if the PRIVMSG in question is an ACTION

`supybot.ircmsgs.isCtcp(msg)`  
Returns whether or not msg is a CTCP message.

`supybot.ircmsgs.join(channel, key=None, prefix="", msg=None)`  
Returns a JOIN to a channel

`supybot.ircmsgs.joins(channels, keys=None, prefix="", msg=None)`  
Returns a JOIN to each of channels.

`supybot.ircmsgs.kick(channel, nick, s="", prefix="", msg=None)`  
Returns a KICK to kick nick from channel with the message s.

`supybot.ircmsgs.kicks(channels, nicks, s="", prefix="", msg=None)`  
Returns a KICK to kick each of nicks from channel with the message s.

`supybot.ircmsgs.modes(channel, args=(), prefix="", msg=None)`  
Returns a MODE message for the channel for all the (mode, targetOrNone) 2-tuples in 'args'.

`supybot.ircmsgs.nick(nick, prefix="", msg=None)`  
Returns a NICK with nick nick.

`supybot.ircmsgs.notice(recipient, s, prefix="", msg=None)`  
Returns a NOTICE to recipient with the message s.

`supybot.ircmsgs.op(channel, nick, prefix="", msg=None)`  
Returns a MODE to op nick on channel.

`supybot.ircmsgs.ops(channel, nicks, prefix="", msg=None)`  
Returns a MODE to op each of nicks on channel.

`supybot.ircmsgs.part(channel, s="", prefix="", msg=None)`  
Returns a PART from channel with the message s.

`supybot.ircmsgs.parts(channels, s="", prefix="", msg=None)`  
Returns a PART from each of channels with the message s.

`supybot.ircmsgs.password(password, prefix="", msg=None)`  
Returns a PASS command for accessing a server.

`supybot.ircmsgs.ping(payload, prefix="", msg=None)`  
Takes a payload and returns the proper PING IrcMsg.

`supybot.ircmsgs.pong(payload, prefix="", msg=None)`  
Takes a payload and returns the proper PONG IrcMsg.

`supybot.ircmsgs.prettyPrint(msg, addRecipients=False, timestampFormat=None, showNick=True)`  
Provides a client-friendly string form for messages.  
IIRC, I copied BitchX's (or was it XChat's?) format for messages.

`supybot.ircmsgs.privmsg(recipient, s, prefix="", msg=None)`  
Returns a PRIVMSG to recipient with the message s.

`supybot.ircmsgs.quit` (*s*="", *prefix*="", *msg=None*)  
Returns a QUIT with the message *s*.

`supybot.ircmsgs.split_args` (*s*, *maxsplit=-1*)  
Splits on spaces, treating consecutive spaces as one.

`supybot.ircmsgs.topic` (*channel*, *topic=None*, *prefix*="", *msg=None*)  
Returns a TOPIC for channel with the topic *topic*.

`supybot.ircmsgs.unAction` (*msg*)  
Returns the payload (i.e., non-ACTION text) of an ACTION msg.

`supybot.ircmsgs.unban` (*channel*, *hostmask*, *prefix*="", *msg=None*)  
Returns a MODE to unban nick on channel.

`supybot.ircmsgs.unbans` (*channel*, *hostmasks*, *prefix*="", *msg=None*)  
Returns a MODE to unban each of nicks on channel.

`supybot.ircmsgs.user` (*ident*, *user*, *prefix*="", *msg=None*)  
Returns a USER with ident *ident* and user *user*.

`supybot.ircmsgs.voice` (*channel*, *nick*, *prefix*="", *msg=None*)  
Returns a MODE to voice nick on channel.

`supybot.ircmsgs.voices` (*channel*, *nicks*, *prefix*="", *msg=None*)  
Returns a MODE to voice each of nicks on channel.

`supybot.ircmsgs.who` (*hostmaskOrChannel*, *prefix*="", *msg=None*, *args=()*)  
Returns a WHO for the hostmask or channel *hostmaskOrChannel*.

`supybot.ircmsgs.whois` (*nick*, *mask*="", *prefix*="", *msg=None*)  
Returns a WHOIS for nick.

`supybot.ircmsgs.whowas` (*nick*, *mask*="", *prefix*="", *msg=None*)  
Returns a WHOIS for nick.

## 2.14.4 supybot.irclib

### Irc

It is usually the *irc* object given to plugin commands.

**class** `supybot.irclib.Irc` (*network*, *callbacks=[]*)  
Bases: `supybot.irclib.IrcCommandDispatcher`, `supybot.log.Firewalled`  
The base class for an IRC connection.  
Handles PING commands already.

**zombie**  
Whether or not this object represents a living IRC connection.  
**Type** bool

**network**  
The name of the network this object is connected to.  
**Type** str

**startedAt**  
When this connection was (re)started.  
**Type** float

**callbacks**

List of all callbacks (ie. plugins) currently loaded

**Type** `List[IrcCallback]`

**queue**

Queue of messages waiting to be sent. Plugins should use the `queueMsg` method instead of accessing this directly.

**Type** `IrcMsgQueue`

**fastqueue**

Same as `queue`, but for messages with high priority. Plugins should use the `sendMsg` method instead of accessing this directly (or `queueMsg` if the message isn't high priority).

**Type** `smallqueue`

**driver**

Driver of the IRC connection (normally, a `supybot.drivers.Socket.SocketDriver` object). Plugins normally do not need to access this.

**startedSync**

When joining a channel, a `'#channel': time.time()` entry is added to this dict, which is then removed when the join is completed. Plugins should not change this value, it is automatically handled when they send a JOIN.

**Type** `ircutils.IrcDict[str, float]`

**monitoring**

A dict with nicks as keys and the number of plugins monitoring this nick as value. Plugins should not access this directly, and should use the `monitor` and `unmonitor` methods instead.

**Type** `ircutils.IrcDict[str, int]`

**state**

An `supybot.irclib.IrcState` object, which stores all the known information about the connection with the IRC network.

**Type** `supybot.irclib.IrcState`

**REQUEST\_CAPABILITIES** = `{'account-notify', 'account-tag', 'away-notify', 'batch', 'chgh`  
IRCv3 capabilities requested when they are available.

echo-message is special-cased to be requested only with labeled-response.

To check if a capability was negotiated, use `irc.state.capabilities_ack`.

**REQUEST\_EXPERIMENTAL\_CAPABILITIES** = `{'draft/account-registration', 'draft/multiline'}`  
Like **REQUEST\_CAPABILITIES**, but these capabilities are only requested if `supybot.protocols.irc.experimentalExtensions` is enabled.

**addCallback** (*callback*)

Adds a callback to the callbacks list.

**Parameters** **callback** (`supybot.irclib.IrcCallback`) – A callback object

**capUpkeep** (*msg*)

Called after getting a CAP ACK/NAK to check it's consistent with what was requested, and to end the cap negotiation when we received all the ACK/NAKs we were waiting for.

*msg* is the message that triggered this call.

**die** (\*args, \*\*kwargs)  
Makes the Irc object *promise* to die – but it won't die (of its own volition) until all its queues are clear.  
Isn't that cool?

**dispatchCommand** (command, args=None)  
Given a string 'command', dispatches to doCommand.

**do002** (msg)  
Logs the ircd version.

**doError** (msg)  
Handles ERROR messages.

**doNick** (msg)  
Handles NICK messages.

**doPing** (msg)  
Handles PING messages.

**doPong** (msg)  
Handles PONG messages.

**feedMsg** (\*args, \*\*kwargs)  
Called by the IrcDriver; feeds a message received.  
  
*tag=False* is used when simulating echo messages, to skip adding received\* tags.

**getCallback** (name)  
Gets a given callback by name.

**isChannel** (s)  
Helper function to check whether a given string is a channel on the network this Irc object is connected to.

**isNick** (s)  
Returns whether the given argument is a valid nick on this network.

**monitor** (targets)  
Increment a counter of how many callbacks monitor each target; and send a MONITOR + to the server if the target is not yet monitored.

**queueBatch** (msgs)  
Queues a batch of messages to be sent to the server. See <<https://ircv3.net/specs/extensions/batch-3.2>>  
  
queueMsg/sendMsg must not be used repeatedly to send a batch, because they do not guarantee the batch is send atomically, which is required because "Clients MUST NOT send messages other than PRIVMSG while a multiline batch is open." – <<https://ircv3.net/specs/extensions/multiline>>

**queueMsg** (msg)  
Queues a message to be sent to the server.

**removeCallback** (name)  
Removes a callback from the callback list.

**requestCapabilities** (caps)  
Takes an iterable of IRCv3 capabilities, and requests them to the server using CAP REQ.  
  
This is mostly just used during connection registration or when the server sends CAP NEW; but plugins may use it as well to request custom capabilities. They should make sure these capabilities cannot negatively impact other plugins, though.

**reset** ()  
Resets the Irc object. Called when the driver reconnects.

**sendMsg** (*msg*)

Queues a message to be sent to the server *immediately*

**takeMsg** (*\*args, \*\*kwargs*)

Called by the IrcDriver; takes a message to be sent.

**unmonitor** (*targets*)

Decrements a counter of how many callbacks monitor each target; and send a MONITOR - to the server if the counter drops to 0.

## IrcState

Used mainly as the state attribute of `supybot.irclib.Irc` objects.

**class** `supybot.irclib.IrcState` (*history=None, supported=None, nicksToHostmasks=None, channels=None, capabilities\_req=None, capabilities\_ack=None, capabilities\_nak=None, capabilities\_ls=None*)

Bases: `supybot.irclib.IrcCommandDispatcher`, `supybot.log.Firewalled`

Maintains state of the Irc connection. Should also become smarter.

**fsm**

A finite-state machine representing the current state of the IRC connection: various steps while connecting, then remains in the CONNECTED state (or CONNECTED\_SASL when doing SASL in the middle of a connection).

**Type** `IrcStateFsm`

**capabilities\_req**

Set of all capabilities requested from the server. See <<https://ircv3.net/specs/core/capability-negotiation>>

**Type** `Set[str]`

**capabilities\_ack**

Set of all capabilities requested from and acknowledged by the server. See <<https://ircv3.net/specs/core/capability-negotiation>>

**Type** `Set[str]`

**capabilities\_nak**

Set of all capabilities requested from and refused by the server. This should always be empty unless the bot, a plugin, or the server is misbehaving. See <<https://ircv3.net/specs/core/capability-negotiation>>

**Type** `Set[str]`

**capabilities\_ls**

Stores all the capabilities advertised by the server, as well as their value, if any.

**Type** `Dict[str, Optional[str]]`

**ircd**

Identification string of the software running the server we are connected to. See <<https://defs.ircdocs.horse/defs/numerics.html#rpl-myinfo-004>>

**Type** `str`

**supported**

Stores the value of ISUPPORT sent when connecting. See <<https://defs.ircdocs.horse/defs/isupport.html>> for the list of keys.

**Type** `utils.InsensitivePreservingDict[str, Any]`

**history**

History of messages received from the network. Automatically discards messages so it doesn't exceed `supybot.protocols.irc.maxHistoryLength`.

**Type** `RingBuffer[ircmsgs.IrcMsg]`

**channels**

Store channel states.

**Type** `ircutils.IrcDict[str, ChannelState]`

**nicksToHostmasks**

Stores the last hostmask of a seen nick.

**Type** `ircutils.IrcDict[str, str]`

**addMsg** (*\*args, \*\*kwargs*)

Updates the state based on the irc object and the message.

**dispatchCommand** (*command, args=None*)

Given a string 'command', dispatches to doCommand.

**do004** (*irc, msg*)

Handles parsing the 004 reply

Supported user and channel modes are cached

**getClientTagDenied** (*tag*)

Returns whether the given tag is denied by the server, according to its CLIENTTAGDENY policy. This is only informative, and servers may still allow or deny tags at their discretion.

For details, see the RPL\_ISUPPORT section in <<https://ircv3.net/specs/extensions/message-tags>>

**getParentBatches** (*msg*)

Given an IrcMsg, returns a list of all batches that contain it, innermost first.

Raises ValueError if msg is not in a batch; or if it is in a batch that has already ended. This restriction may be relaxed in the future.

This means that you should not call `getParentBatches` on a message that was already processed.

For example, assume Limnoria received the following:

```
:irc.host BATCH +outer example.com/foo
@batch=outer :irc.host BATCH +inner example.com/bar
@batch=inner :nick!user@host PRIVMSG #channel :Hi
@batch=outer :irc.host BATCH -inner
:irc.host BATCH -outer
```

If you call `getParentBatches` on any of the middle three messages, you get `[Batch(name='inner', ...), Batch(name='outer', ...)]`. And if you call `getParentBatches` on either the first or the last message, you get `[Batch(name='outer', ...)]`

And you may only call `getParentBatches` on the PRIVMSG if only the first three messages were processed.

**getTopic** (*channel*)

Returns the topic for a given channel.

**nickToHostmask** (*nick*)

Returns the hostmask for a given nick.

**reset** ()

Resets the state to normal, unconnected state.

## IrcStateFsm

Used as the `fsm` attribute of `supybot.irclib.IrcState` objects

```
class supybot.irclib.IrcStateFsm
```

Bases: `object`

Finite State Machine keeping track of what part of the connection initialization we are in.

```
class States
```

Bases: `enum.Enum`

Enumeration of all the states of an IRC connection.

```
CONNECTED = 70
```

Normal state of the connections

```
CONNECTED_SASL = 80
```

Doing SASL authentication in the middle of a connection.

```
INIT_CAP_NEGOTIATION = 20
```

Sent CAP LS, did not send CAP END yet

```
INIT_MOTD = 60
```

Waiting for end of MOTD

```
INIT_SASL = 30
```

In an AUTHENTICATE session

```
INIT_WAITING_MOTD = 50
```

Waiting for start of MOTD

```
UNINITIALIZED = 10
```

Nothing received yet (except server notices)

```
on_cap_end(irc, msg)
```

When we send CAP END

```
on_end_motd(irc, msg)
```

On 376 (RPL\_ENDOFMOTD) or 422 (ERR\_NOMOTD)

```
on_init_messages_sent(irc)
```

As soon as USER/NICK/CAP LS are sent

```
on_sasl_auth_finished(irc, msg)
```

When sasl auth either succeeded or failed.

```
on_sasl_cap(irc, msg)
```

Whenever we see the 'sasl' capability in a CAP LS response

```
on_start_motd(irc, msg)
```

On 375 (RPL\_MOTDSTART)

## ChannelState

Used mainly as the `.state.channels['#chan']` attribute of `supybot.irclib.Irc` objects.

```
class supybot.irclib.ChannelState
```

Bases: `supybot.utils.python.Object`

Represents the known state of an IRC channel.

**topic**

The topic of a channel (possibly the empty string)

**Type** str

**created**

Timestamp of the channel creation, according to the server.

**Type** int

**ops**

Set of the nicks of all the operators of the channel.

**Type** ircutils.IrcSet[str]

**halfops**

Set of the nicks of all the half-operators of the channel.

**Type** ircutils.IrcSet[str]

**voices**

Set of the nicks of all the voiced users of the channel.

**Type** ircutils.IrcSet[str]

**users**

Set of the nicks of all the users in the channel.

**Type** ircutils.IrcSet[str]

**bans**

Set of the all the banmasks set in the channel.

**Type** ircutils.IrcSet[str]

**modes**

Dict of all the modes set in the channel, with they value, if any. This excludes the following modes: ovhbeq

**Type** Dict[str, Optional[str]]

**addUser** (*user*, *prefix\_chars*= '@%+&~!')

Adds a given user to the ChannelState. Power prefixes are handled.

**isHalfop** (*nick*)

Returns whether the given nick is an halfop.

**isHalfopPlus** (*nick*)

Returns whether the given nick is an halfop, or an op.

**isOp** (*nick*)

Returns whether the given nick is an op.

**isOpPlus** (*nick*)

Returns whether the given nick is an op.

**isVoice** (*nick*)

Returns whether the given nick is voiced.

**isVoicePlus** (*nick*)

Returns whether the given nick is voiced, an halfop, or an op.

**removeUser** (*user*)

Removes a given user from the channel.

**replaceUser** (*oldNick*, *newNick*)

Changes the user oldNick to newNick; used for NICK changes.



## Other classes

**class** `supybot.irclib.Batch` (*name, type, arguments, messages, parent\_batch*)

Bases: `tuple`

Represents a batch of messages, see <<https://ircv3.net/specs/extensions/batch-3.2>>

Only access attributes by their name and do not create Batch objects in plugins; so we can extend the structure without breaking plugins.

### arguments

Alias for field number 2

### messages

Alias for field number 3

### name

Alias for field number 0

### parent\_batch

Alias for field number 4

### type

Alias for field number 1

**class** `supybot.irclib.IrcCallback` (*\*args, \*\*kwargs*)

Bases: `supybot.irclib.IrcCommandDispatcher`, `supybot.log.Firewalled`

Base class for standard callbacks.

Callbacks derived from this class should have methods of the form “doCommand” – doPrivmsg, doNick, do433, etc. These will be called on matching messages.

**callPrecedence** (*\*args, \*\*kwargs*)

Returns a pair of (callbacks to call before me, callbacks to call after me)

**die** (*\*args, \*\*kwargs*)

Makes the callback die. Called when the parent Irc object dies.

**inFilter** (*\*args, \*\*kwargs*)

Used for filtering/modifying messages as they’re entering.

`ircmsgs.IrcMsg` objects are immutable, so this method is expected to return another `ircmsgs.IrcMsg` object. Obviously the same `IrcMsg` can be returned.

**name** (*\*args, \*\*kwargs*)

Returns the name of the callback.

**outFilter** (*\*args, \*\*kwargs*)

Used for filtering/modifying messages as they’re leaving.

As with `inFilter`, an `IrcMsg` is returned.

**postTransition** (*\*args, \*\*kwargs*)

Called when the state of the IRC connection changes.

*msg* is the message that triggered the transition, if any.

**reset** (*\*args, \*\*kwargs*)

Resets the callback. Called when reconnecting to the server.

**class** `supybot.irclib.IrcCommandDispatcher`

Bases: `object`

Base class for classes that must dispatch on a command.

**dispatchCommand** (*command*, *args=None*)

Given a string 'command', dispatches to doCommand.

**class** supybot.irclib.IrcMsgQueue (*iterable=()*)

Bases: object

Class for a queue of IrcMsgs. Eventually, it should be smart.

Probably smarter than it is now, though it's gotten quite a bit smarter than it originally was. A method to "score" methods, and a heapq to maintain a priority queue of the messages would be the ideal way to do intelligent queuing.

As it stands, however, we simply keep track of 'high priority' messages, 'low priority' messages, and normal messages, and just make sure to return the 'high priority' ones before the normal ones before the 'low priority' ones.

**dequeue** ()

Dequeues a given message.

**enqueue** (*msg*)

Enqueues a given message.

**reset** ()

Clears the queue.

## 2.14.5 supybot.registry

**class** supybot.registry.Boolean (*default*, *help*, *setDefault=True*, *showDefault=True*, *\*\*kwargs*)

Bases: [supybot.registry.Value](#)

Value must be either True or False (or On or Off).

**addCallback** (*callback*, *\*args*, *\*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None*, *channel=None*, *check=True*, *fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call *self.setValue* to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a *IntegerLessThanOneHundred* (all your values must be integers less than 100) convert to an integer in *set()* and check that the integer is less than 100 in this method. You *must* call this parent method in your own *setValue*.

```
class supybot.registry.CommaSeparatedListOfStrings (default, help, setDefault=True,
                                                    showDefault=True, **kwargs)
```

Bases: `supybot.registry.SeparatedListOf`

**List**  
alias of `builtins.list`

**Value**  
alias of `String`

**addCallback** (*callback*, \*args, \*\*kwargs)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**joiner** ()

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**splitter** (*s*)

Override this with a function that takes a string and returns a list of strings.

```
class supybot.registry.CommaSeparatedSetOfStrings (default, help, setDefault=True,
                                                    showDefault=True, **kwargs)
```

Bases: `supybot.registry.SeparatedListOf`

**List**  
alias of `builtins.set`

**Value**  
alias of `String`

**addCallback** (*callback*, \*args, \*\*kwargs)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**joiner** ()

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**splitter** (*s*)

Override this with a function that takes a string and returns a list of strings.

**class** `supybot.registry.Float` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.Value`

Value must be a floating-point number.

**addCallback** (*callback*, \*args, \*\*kwargs)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing

bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**class** `supybot.registry.Group` (*help=""*, *orderAlphabetically=True*, *private=False*)

Bases: `object`

A group; it doesn't hold a value unless handled by a subclass.

**class** `supybot.registry.Integer` (*default*, *help*, *setDefault=True*, *showDefault=True*, *\*\*kwargs*)

Bases: `supybot.registry.Value`

Value must be an integer.

**addCallback** (*callback*, *\*args*, *\*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None*, *channel=None*, *check=True*, *fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**exception** `supybot.registry.InvalidRegistryFile`

Bases: `supybot.registry.RegistryException`

**with\_traceback** ()

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

**exception** `supybot.registry.InvalidRegistryName`

Bases: `supybot.registry.RegistryException`

**with\_traceback** ()

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**exception** `supybot.registry.InvalidRegistryValue`

Bases: `supybot.registry.RegistryException`

**with\_traceback** ()

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** `supybot.registry.Json` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.String`

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**editable** ()

Return an editable dict usable within a ‘with’ statement and committed to the configuration variable at the end.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*v*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you’re a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**exception** `supybot.registry.NonExistentRegistryEntry`

Bases: `supybot.registry.RegistryException`, `AttributeError`

**with\_traceback** ()

Exception.with\_traceback(tb) – set self.\_\_traceback\_\_ to tb and return self.

**class** `supybot.registry.NonNegativeInteger` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.Integer`

Value must be a non-negative integer.

**addCallback** (*callback*, \*args, \*\*kwargs)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call *self.setValue* to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a *IntegerLessThanOneHundred* (all your values must be integers less than 100) convert to an integer in *set()* and check that the integer is less than 100 in this method. You *must* call this parent method in your own *setValue*.

**class** *supybot.registry.NormalizedString* (*default*, \*args, \*\*kwargs)

Bases: *supybot.registry.String*

**addCallback** (*callback*, \*args, \*\*kwargs)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call *self.setValue* to set the value.

**setValue** (*s*)

Check conditions on the actual value type here. I.e., if you're a *IntegerLessThanOneHundred* (all your

values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**class** `supybot.registry.OnlySomeStrings` (*\*args, \*\*kwargs*)

Bases: `supybot.registry.String`

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*s*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**class** `supybot.registry.PositiveFloat` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.Float`

Value must be a floating-point number greater than zero.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.



**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**class** `supybot.registry.PositiveInteger` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.NonNegativeInteger`

Value must be positive (non-zero) integer.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**class** `supybot.registry.Probability` (*\*args, \*\*kwargs*)

Bases: `supybot.registry.Float`

Value must be a floating point number in the range [0, 1].

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**class** `supybot.registry.Regexp` (*default, \*args, \*\*kwargs*)

Bases: `supybot.registry.Value`

Value must be a valid regular expression.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this `Value`. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this `Value` is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**exception** `supybot.registry.RegistryException`

Bases: `Exception`

**with\_traceback** ()

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

**class** `supybot.registry.SeparatedListOf` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.Value`

**List**

alias of `builtins.list`

**class Value** (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.Group`

Invalid registry value. If you're getting this message, report it, because we forgot to put a proper help string here.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**joiner** (*L*)

Override this to join the internal list for output.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**splitter** (*s*)

Override this with a function that takes a string and returns a list of strings.

**class** `supybot.registry.SpaceSeparatedListOf` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.SeparatedListOf`

**List**

alias of `builtins.list`

**class** `Value` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.Group`

Invalid registry value. If you're getting this message, report it, because we forgot to put a proper help string here.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this `Value`. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this `Value` is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**joiner** ()

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `‘.’.join(['ab', 'pq', 'rs']) -> ‘ab.pq.rs’`

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**splitter** (*s*)

Override this with a function that takes a string and returns a list of strings.

**class** `supybot.registry.SpaceSeparatedListOfStrings` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.SpaceSeparatedListOf`

**List**

alias of `builtins.list`

**Value**

alias of `String`

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing

bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**joiner()**

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**splitter** (*s*)

Override this with a function that takes a string and returns a list of strings.

**class** `supybot.registry.SpaceSeparatedSetOfStrings` (*default*, *help*, *setDefault=True*,  
*showDefault=True*, *\*\*kwargs*)

Bases: `supybot.registry.SpaceSeparatedListOfStrings`

**List**

alias of `builtins.set`

**Value**

alias of `String`

**addCallback** (*callback*, *\*args*, *\*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None*, *channel=None*, *check=True*, *fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**joiner()**

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**splitter** (*s*)

Override this with a function that takes a string and returns a list of strings.

**class** `supybot.registry.String` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.Value`

Value is not a valid Python string.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If `check=True` (the default), this will raise an error if `network` (resp. `channel`) is provided but this Value is not network-specific (resp. channel-specific). If `check=False`, then `network` and/or `channel` may be silently ignored.

If `fallback_to_channel=True` (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call `self.setValue` to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a `IntegerLessThanOneHundred` (all your values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

**class** `supybot.registry.StringSurroundedBySpaces` (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: `supybot.registry.String`

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.



**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call *self.setValue* to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a *IntegerLessThanOneHundred* (all your values must be integers less than 100) convert to an integer in *set()* and check that the integer is less than 100 in this method. You *must* call this parent method in your own *setValue*.

**class** *supybot.registry.StringWithSpaceOnRight* (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: *supybot.registry.String*

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call *self.setValue* to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a *IntegerLessThanOneHundred* (all your values must be integers less than 100) convert to an integer in *set()* and check that the integer is less than 100 in this method. You *must* call this parent method in your own *setValue*.

**class** *supybot.registry.TemplatedString* (*\*args, \*\*kwargs*)

Bases: *supybot.registry.String*

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can



give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call *self.setValue* to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a *IntegerLessThanOneHundred* (all your values must be integers less than 100) convert to an integer in *set()* and check that the integer is less than 100 in this method. You *must* call this parent method in your own *setValue*.

**class** *supybot.registry.Value* (*default, help, setDefault=True, showDefault=True, \*\*kwargs*)

Bases: *supybot.registry.Group*

Invalid registry value. If you're getting this message, report it, because we forgot to put a proper help string here.

**addCallback** (*callback, \*args, \*\*kwargs*)

Add a callback to the list. A callback is a function that will be called when the value is changed. You can give this function as many extra arguments as you wish, they will be passed to the callback.

**context** (*value*)

Return a context manager object, which sets this variable to a temporary value, and set the previous value back when exiting the context.

**getSpecific** (*network=None, channel=None, check=True, fallback\_to\_channel=True*)

Gets the network-specific and/or channel-specific value of this Value. If *check=True* (the default), this will raise an error if *network* (resp. *channel*) is provided but this Value is not network-specific (resp. channel-specific). If *check=False*, then *network* and/or *channel* may be silently ignored.

If *fallback\_to\_channel=True* (the default) and the network-specific + channel-specific value is not set, but the channel-specific value is set, it will return the latter. This is useful to upgrade from existing bot configuration, that did not support network-specific values; but it may be undesirable when setting new values.

**removeCallback** (*callback*)

Remove all occurrences of this callbacks from the callback list.

**set** (*s*)

Override this with a function to convert a string to whatever type you want, and call *self.setValue* to set the value.

**setValue** (*v*)

Check conditions on the actual value type here. I.e., if you're a *IntegerLessThanOneHundred* (all your

values must be integers less than 100) convert to an integer in `set()` and check that the integer is less than 100 in this method. You *must* call this parent method in your own `setValue`.

`supybot.registry.open_registry(filename, clear=False)`

Initializes the module by loading the registry file into memory.

## 3.1 Contributing to Limnoria as a developer

---

**Note:** This page is still a draft and is not complete.

---

### 3.1.1 About the policy about repository access

For the moment, I decided to give write access to my repository to nobody, because I want to check everything that is pushed in it. If someone pushes a bad update, it may be dangerous for users and I do not want that.

On the other side, I am **very** open to pull requests, that's to say, if you ask me to merge some changes you made, there are 99% changes I will merge this changes. That's why I suggest you to fork my repository on GitHub, make your modifications, and click the "Pull requests" button in my repository.

### 3.1.2 Using Git

If you are a developer, I assume that you know how to use Git. If you don't, I suggest you to learn how to use it, at least the basics (clone, checkout, branch, commit, push/pull, add/rm, log, show, reset, revert).

I learnt how to use Git with an ebook, also available as a real book: [Pro Git](#).

Our preferred way of contributing is through GitHub pull requests to [Limnoria's repository](#). Please send your pull requests to the *testing* branch.

### 3.1.3 Where to start

If you are not an experienced Python and/or Limnoria developer, you can start with solving [issues tagged as easy](#). I believe they are likely to be easy to solve even without a lot of experience.

If you need help solving an issue (tagged as easy or not) or want to find an issue that matches your skills, please ask on IRC, we will be glad to help you.

### 3.1.4 Code style

Read the doc in the source code (docs/STYLE.rst).

## 3.2 Translating Limnoria

I already wrote a [guide on how to translate plugins](#). So, this page will only explain how to translate the core and push your translations to Limnoria.

### 3.2.1 The best way: using Git yourself

As I said in the [policy about developer's contributions](#), I don't give write access to my repo for the moment, but I accept pull requests.

As you are a translator, you don't need to know all the technical details about development, so I write a simplified doc here.

#### Preparing git

First you should install git. It's usually package git in your OS, or you can download it from [their homepage](#) or download GitHub client for [Windows](#) or [OS X](#)

Then you should tell GitHub who you are and what is your email address. This information is attached to commits and GitHub uses it to get your gravatar:

```
git config --global user.name "Real Name or Nickname here"
git config --global user.email "someone@example.com"
```

If you are going to use the https, you probably want git to remember your GitHub password for some time so you don't have to write it continuously:

```
git config --global credential.helper cache
git config --global credential.helper "cache --timeout=3600"
```

This would make git remember your password for hour. It can be changed by changing 3600 to any other amount of seconds.

#### Cloning the repository

You first need an account on [GitHub](#); I think you don't need explanation for that.

Then, go on [Limnoria repository](#) and click the *Fork* button. This will create you a copy of my repository where you will have write access (and I won't have this write access).

Then, open a console, and write (replace *YourName* by the name of your GitHub account):

```
git clone https://github.com/<YourName>/Limnoria.git --branch=testing
```

If you are experienced with git, you can `git clone git@github.com:<YourName>/Limnoria.git --branch=testing` instead.

This will create a new directory, called *Limnoria*, where all the code and project history are copied. Now, `cd` to the directory:

```
cd Limnoria/
```

The things below affect to you only if you didn't specify the branch in the git clone command.

Then, you need to checkout the *testing* branch. What does that mean? It means that there is different stages in Limnoria: all changes are made in testing, and when I think *testing* is stable, I merge it into *master*. So, checking out *testing* means Git will use the code in *testing*, you will translate strings that are in *testing*, and changes you make will be in *testing*. Now, do it:

```
git checkout testing
```

Git will reply you that it understood what you mean by *testing*.

Ok, now, you can translate.

## Pushing translations

Once you have done some translations (let's say you translated Alias), you have to commit your changes. That means you tell Git "Ok, I've made some changes, and I want to take a snapshot (either to be able to roll back or to publish your changes).

First, you need to tell Git what files you want to be committed (let's say you are the Finnish translator, so you updated Alias's fi.po):

```
git add plugins/Alias/locales/fi.po
```

Then, you can commit your files. Depending on what you made, you can use one of these commands (not all of them!):

```
git commit -m "Alias: Add l10n-fi."
git commit -m "Alias: Update l10n-fi."
git commit -m "Alias: Fix l10n-fi."
```

By the way, the text that follows -m is a message that will be read by **humans**, so you can write anything you want, but I think it's better that everybody use the same kind of messages.

Ok, then, Git knows you have done something. But you didn't send your work on Internet yet. To send it, run:

```
git push
```

Simple, isn't it?

Now, go back to GitHub and your forked repository, and click the *Pull request* button. Then, set *testing* on the both side, and run *Update Commit Range*. I will be mailed that you asked me to merge your changes, and I will do it soon.

## Getting updates

As you may know, I do some updates in Limnoria repository. ;)

You need to have the latest version of the *messages.pot* files. So, you need to teach Git how to get these updates:

```
git remote add upstream https://github.com/ProgVal/Limnoria.git
```

Now, every time you want to download updates, run:

```
git fetch upstream
git merge upstream/testing
```

### 3.2.2 Another way: mailing me your translations

I think this is the simplest way for you. You only have to follow the translation guide and send me your .po files by mail.

You can choose either one of this way to do it.

#### Mikaela's way

Send the fi.po (or whatever the name is) files one by one as an attachment. Don't forget to tell me what plugin it is.

I (Mikaela) have moved to git long time ago though.

#### skizzhg's way

Do many translations. Put them in a tarball/zipball/whatever (but not a RAR archive, I can't read them because is a proprietary format).

I prefer that you choose this architecture:

- FirstPlugin/locales/it.po
- SecondPlugin/locales/it.po
- ThirdPlugin/locales/it.po

Because I can extract everything with one click.

---

**Note:** This page is a work in progress, and is still very incomplete. Please send your ideas/suggestions on the IRC channel!

---

**boolean** A value that can be either True or False.

**command** An action that can be triggered by typing its name on IRC.

**configuration variable**

**configuration value** A configuration variable is an object with a name that can be set to different values to change the behavior of the bot.

They can be changed with the plugin-Config plugin.

**inFilter** Some code that replaces messages right after the bot receives them from IRC, and before it starts processing them.

This is the opposite of *outFilter*.

**network** An IRC network, ie. a group of connected IRC servers, that share the same set of channels and users

**outfilter**

**outFilter** Some code or command that replaces messages just before the bot sends them to IRC.

Some plugins define them for their own purposes, such as plugin-ShrinkUrl to replace URLs. The plugin-Filter plugin provides an `outfilter` command to allow bot admins to customize the messages written by their bot.

This is the opposite of *inFilter*.

**plugin** Some Python code/package that provides commands.

**server** A node in an IRC :term‘network’. Limnoria usually does not care about servers, and deals with entire networks as a single entity.

**specific**

**channel-specific**

**network-specific** A *configuration variable* is said to be channel-specific and/or network-specific when it can take different values depending on the channel/network it is used in.



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### S

`supybot.callbacks`, [84](#)  
`supybot.commands`, [89](#)  
`supybot.irclib`, [101](#)  
`supybot.ircmsgs`, [91](#)  
`supybot.registry`, [102](#)



## A

action() (in module *supybot.ircmsgs*), 92  
 add() (*supybot.callbacks.DisabledCommands* method), 86  
 addCallback() (*supybot.irclib.Irc* method), 95  
 addCallback() (*supybot.registry.Boolean* method), 102  
 addCallback() (*supybot.registry.CommaSeparatedListOfStrings* method), 103  
 addCallback() (*supybot.registry.CommaSeparatedSetOfStrings* method), 103  
 addCallback() (*supybot.registry.Float* method), 104  
 addCallback() (*supybot.registry.Integer* method), 105  
 addCallback() (*supybot.registry.Json* method), 106  
 addCallback() (*supybot.registry.NonNegativeInteger* method), 106  
 addCallback() (*supybot.registry.NormalizedString* method), 107  
 addCallback() (*supybot.registry.OnlySomeStrings* method), 108  
 addCallback() (*supybot.registry.PositiveFloat* method), 108  
 addCallback() (*supybot.registry.PositiveInteger* method), 109  
 addCallback() (*supybot.registry.Probability* method), 109  
 addCallback() (*supybot.registry.Regexp* method), 110  
 addCallback() (*supybot.registry.SeparatedListOf* method), 111  
 addCallback() (*supybot.registry.SeparatedListOf.Value* method), 111  
 addCallback() (*supybot.registry.SpaceSeparatedList* method), 112

addCallback() (*supybot.registry.SpaceSeparatedListOf.Value* method), 112  
 addCallback() (*supybot.registry.SpaceSeparatedListOfStrings* method), 113  
 addCallback() (*supybot.registry.SpaceSeparatedSetOfStrings* method), 114  
 addCallback() (*supybot.registry.String* method), 115  
 addCallback() (*supybot.registry.StringSurroundedBySpaces* method), 115  
 addCallback() (*supybot.registry.StringWithSpaceOnRight* method), 116  
 addCallback() (*supybot.registry.TemplatedString* method), 116  
 addCallback() (*supybot.registry.Value* method), 117  
 addConverter() (in module *supybot.commands*), 90  
 additional (class in *supybot.commands*), 90  
 addMsg() (*supybot.irclib.IrcState* method), 98  
 addressed() (in module *supybot.callbacks*), 89  
 addressedRegexp (*supybot.callbacks.PluginRegexp* attribute), 84  
 addUser() (*supybot.irclib.ChannelState* method), 100  
 alwaysCall (*supybot.callbacks.PluginMixin* attribute), 87  
 any (class in *supybot.commands*), 89  
 args (*supybot.ircmsgs.IrcMsg* attribute), 91  
 ArgumentError, 84  
 arguments (*supybot.irclib.Batch* attribute), 101

## B

ban() (in module *supybot.ircmsgs*), 92  
 bans (*supybot.irclib.ChannelState* attribute), 100  
 bans() (in module *supybot.ircmsgs*), 92  
 BasePlugin (class in *supybot.callbacks*), 84  
 Batch (class in *supybot.irclib*), 101  
 boolean, 123

Boolean (*class in supybot.registry*), 102

## C

callbacks (*supybot.irclib.Irc attribute*), 94

callCommand() (*supybot.callbacks.Commands method*), 85

callCommand() (*supybot.callbacks.Plugin method*), 83

callConverter() (*in module supybot.commands*), 90

callPrecedence() (*supybot.callbacks.Plugin method*), 83

callPrecedence() (*supybot.irclib.IrcCallback method*), 101

canonicalName() (*in module supybot.callbacks*), 89

canonicalName() (*supybot.callbacks.Commands method*), 85

canonicalName() (*supybot.callbacks.PluginMixin method*), 87

CanonicalNameDict (*class in supybot.callbacks*), 84

CanonicalNameSet (*class in supybot.callbacks*), 85

CanonicalString (*class in supybot.callbacks*), 85

capabilities\_ack (*supybot.irclib.IrcState attribute*), 97

capabilities\_ls (*supybot.irclib.IrcState attribute*), 97

capabilities\_nak (*supybot.irclib.IrcState attribute*), 97

capabilities\_req (*supybot.irclib.IrcState attribute*), 97

capUpkeep() (*supybot.irclib.Irc method*), 95

channel (*supybot.ircmsgs.IrcMsg attribute*), 91

channel-specific, 123

channels (*supybot.irclib.IrcState attribute*), 98

ChannelState (*class in supybot.irclib*), 99

checkCommandCapability() (*in module supybot.callbacks*), 89

classModule (*supybot.callbacks.PluginMixin attribute*), 87

commalist (*class in supybot.commands*), 90

command, 123

command (*supybot.ircmsgs.IrcMsg attribute*), 91

commandArgs (*supybot.callbacks.Commands attribute*), 85

CommandProcess (*class in supybot.callbacks*), 85

Commands (*class in supybot.callbacks*), 85

CommandThread (*class in supybot.callbacks*), 85

CommaSeparatedListOfStrings (*class in supybot.registry*), 103

CommaSeparatedSetOfStrings (*class in supybot.registry*), 103

configuration value, 123

configuration variable, 123

CONNECTED (*supybot.irclib.IrcStateFsm.States attribute*), 99

CONNECTED\_SASL (*supybot.irclib.IrcStateFsm.States attribute*), 99

context (*class in supybot.commands*), 89

context() (*supybot.registry.Boolean method*), 102

context() (*supybot.registry.CommaSeparatedListOfStrings method*), 103

context() (*supybot.registry.CommaSeparatedSetOfStrings method*), 104

context() (*supybot.registry.Float method*), 104

context() (*supybot.registry.Integer method*), 105

context() (*supybot.registry.Json method*), 106

context() (*supybot.registry.NonNegativeInteger method*), 107

context() (*supybot.registry.NormalizedString method*), 107

context() (*supybot.registry.OnlySomeStrings method*), 108

context() (*supybot.registry.PositiveFloat method*), 108

context() (*supybot.registry.PositiveInteger method*), 109

context() (*supybot.registry.Probability method*), 109

context() (*supybot.registry.Regexp method*), 110

context() (*supybot.registry.SeparatedListOf method*), 111

context() (*supybot.registry.SeparatedListOf.Value method*), 111

context() (*supybot.registry.SpaceSeparatedListOf method*), 113

context() (*supybot.registry.SpaceSeparatedListOf.Value method*), 112

context() (*supybot.registry.SpaceSeparatedListOfStrings method*), 113

context() (*supybot.registry.SpaceSeparatedSetOfStrings method*), 114

context() (*supybot.registry.String method*), 115

context() (*supybot.registry.StringSurroundedBySpaces method*), 115

context() (*supybot.registry.StringWithSpaceOnRight method*), 116

context() (*supybot.registry.TemplatedString method*), 117

context() (*supybot.registry.Value method*), 117

created (*supybot.irclib.ChannelState attribute*), 100

## D

dehalfop() (*in module supybot.ircmsgs*), 92

dehalfops() (*in module supybot.ircmsgs*), 92

deop() (*in module supybot.ircmsgs*), 92

deops() (*in module supybot.ircmsgs*), 92

dequeue() (*supybot.irclib.IrcMsgQueue method*), 102

devoice() (*in module supybot.ircmsgs*), 92

devoices() (in module *supybot.ircmsgs*), 92  
 die() (*supybot.callbacks.Plugin* method), 83  
 die() (*supybot.irclib.Irc* method), 95  
 die() (*supybot.irclib.IrcCallback* method), 101  
 Disabled (class in *supybot.callbacks*), 86  
 disabled() (*supybot.callbacks.DisabledCommands* method), 86  
 DisabledCommands (class in *supybot.callbacks*), 86  
 dispatchCommand() (*supybot.callbacks.Plugin* method), 83  
 dispatchCommand() (*supybot.irclib.Irc* method), 96  
 dispatchCommand() (*supybot.irclib.IrcCommandDispatcher* method), 101  
 dispatchCommand() (*supybot.irclib.IrcState* method), 98  
 do002() (*supybot.irclib.Irc* method), 96  
 do004() (*supybot.irclib.IrcState* method), 98  
 doError() (*supybot.irclib.Irc* method), 96  
 doNick() (*supybot.irclib.Irc* method), 96  
 doPing() (*supybot.irclib.Irc* method), 96  
 doPong() (*supybot.irclib.Irc* method), 96  
 driver (*supybot.irclib.Irc* attribute), 95

## E

editable() (*supybot.registry.Json* method), 106  
 enqueue() (*supybot.irclib.IrcMsgQueue* method), 102  
 Error, 86  
 error() (in module *supybot.callbacks*), 89  
 error() (*supybot.callbacks.NestedCommandsIrcProxy* method), 86  
 error() (*supybot.callbacks.ReplyIrcProxy* method), 88  
 errorInvalid() (*supybot.callbacks.RichReplyMethods* method), 88  
 errorNoCapability() (*supybot.callbacks.RichReplyMethods* method), 88  
 errorNotRegistered() (*supybot.callbacks.RichReplyMethods* method), 88  
 errorNoUser() (*supybot.callbacks.RichReplyMethods* method), 88  
 errorPossibleBug() (*supybot.callbacks.RichReplyMethods* method), 88  
 errorRequiresPrivacy() (*supybot.callbacks.RichReplyMethods* method), 89  
 evalArgs() (*supybot.callbacks.NestedCommandsIrcProxy* method), 86

## F

fastqueue (*supybot.irclib.Irc* attribute), 95  
 feedMsg() (*supybot.irclib.Irc* method), 96  
 finalEval() (*supybot.callbacks.NestedCommandsIrcProxy* method), 87  
 findCallbacksForArgs() (*supybot.callbacks.NestedCommandsIrcProxy* method), 87  
 first (class in *supybot.commands*), 90  
 Float (class in *supybot.registry*), 104  
 formatCommand() (in module *supybot.callbacks*), 89  
 fsm (*supybot.irclib.IrcState* attribute), 97

## G

getCallback() (*supybot.irclib.Irc* method), 96  
 getClientTagDenied() (*supybot.irclib.IrcState* method), 98  
 getCommand() (*supybot.callbacks.Commands* method), 85  
 getCommand() (*supybot.callbacks.Plugin* method), 83  
 getCommandHelp() (*supybot.callbacks.Commands* method), 85  
 getCommandHelp() (*supybot.callbacks.Plugin* method), 83  
 getCommandMethod() (*supybot.callbacks.Commands* method), 85  
 getCommandMethod() (*supybot.callbacks.Plugin* method), 83  
 getConverter() (in module *supybot.commands*), 90  
 getHelp() (in module *supybot.callbacks*), 89  
 getopts (class in *supybot.commands*), 90  
 getParentBatches() (*supybot.irclib.IrcState* method), 98  
 getPluginHelp() (*supybot.callbacks.PluginMixin* method), 87  
 getRealIrc() (*supybot.callbacks.ReplyIrcProxy* method), 88  
 getSpecific() (*supybot.registry.Boolean* method), 102  
 getSpecific() (*supybot.registry.CommaSeparatedListOfStrings* method), 103  
 getSpecific() (*supybot.registry.CommaSeparatedSetOfStrings* method), 104  
 getSpecific() (*supybot.registry.Float* method), 104  
 getSpecific() (*supybot.registry.Integer* method), 105  
 getSpecific() (*supybot.registry.Json* method), 106  
 getSpecific() (*supybot.registry.NonNegativeInteger* method), 107  
 getSpecific() (*supybot.registry.NormalizedString* method), 107

[getSpecific\(\)](#) (*supybot.registry.OnlySomeStrings method*), 108  
[getSpecific\(\)](#) (*supybot.registry.PositiveFloat method*), 108  
[getSpecific\(\)](#) (*supybot.registry.PositiveInteger method*), 109  
[getSpecific\(\)](#) (*supybot.registry.Probability method*), 109  
[getSpecific\(\)](#) (*supybot.registry.Regexp method*), 110  
[getSpecific\(\)](#) (*supybot.registry.SeparatedListOf method*), 111  
[getSpecific\(\)](#) (*supybot.registry.SeparatedListOf.Value method*), 111  
[getSpecific\(\)](#) (*supybot.registry.SpaceSeparatedListOf method*), 113  
[getSpecific\(\)](#) (*supybot.registry.SpaceSeparatedListOf.Value method*), 112  
[getSpecific\(\)](#) (*supybot.registry.SpaceSeparatedListOfStrings method*), 113  
[getSpecific\(\)](#) (*supybot.registry.SpaceSeparatedSetOfStrings method*), 114  
[getSpecific\(\)](#) (*supybot.registry.String method*), 115  
[getSpecific\(\)](#) (*supybot.registry.StringSurroundedBySpaces method*), 115  
[getSpecific\(\)](#) (*supybot.registry.StringWithSpaceOnRight method*), 116  
[getSpecific\(\)](#) (*supybot.registry.TemplatedString method*), 117  
[getSpecific\(\)](#) (*supybot.registry.Value method*), 117  
[getSyntax\(\)](#) (*in module supybot.callbacks*), 89  
[getTopic\(\)](#) (*supybot.irclib.IrcState method*), 98  
[Group](#) (*class in supybot.registry*), 105

## H

[halfop\(\)](#) (*in module supybot.ircmsgs*), 92  
[halfops](#) (*supybot.irclib.ChannelState attribute*), 100  
[halfops\(\)](#) (*in module supybot.ircmsgs*), 93  
[history](#) (*supybot.irclib.IrcState attribute*), 97  
[host](#) (*supybot.ircmsgs.IrcMsg attribute*), 91

## I

[inFilter](#), 123  
[inFilter\(\)](#) (*supybot.callbacks.Plugin method*), 83  
[inFilter\(\)](#) (*supybot.irclib.IrcCallback method*), 101  
[INIT\\_CAP\\_NEGOTIATION](#) (*supybot.irclib.IrcStateFsm.States attribute*), 99

[INIT\\_MOTD](#) (*supybot.irclib.IrcStateFsm.States attribute*), 99  
[INIT\\_SASL](#) (*supybot.irclib.IrcStateFsm.States attribute*), 99  
[INIT\\_WAITING\\_MOTD](#) (*supybot.irclib.IrcStateFsm.States attribute*), 99  
[Integer](#) (*class in supybot.registry*), 105  
[InvalidRegistryFile](#), 105  
[InvalidRegistryName](#), 105  
[InvalidRegistryValue](#), 106  
[invite\(\)](#) (*in module supybot.ircmsgs*), 93  
[Irc](#) (*class in supybot.irclib*), 94  
[IrcCallback](#) (*class in supybot.irclib*), 101  
[IrcCommandDispatcher](#) (*class in supybot.irclib*), 101  
[ircd](#) (*supybot.irclib.IrcState attribute*), 97  
[IrcMsg](#) (*class in supybot.ircmsgs*), 91  
[IrcMsgQueue](#) (*class in supybot.irclib*), 102  
[IrcObjectProxy](#) (*in module supybot.callbacks*), 86  
[IrcState](#) (*class in supybot.irclib*), 97  
[IrcStateFsm](#) (*class in supybot.irclib*), 99  
[IrcStateFsm.States](#) (*class in supybot.irclib*), 99  
[isAction\(\)](#) (*in module supybot.ircmsgs*), 93  
[isChannel\(\)](#) (*supybot.irclib.Irc method*), 96  
[isCommand\(\)](#) (*supybot.callbacks.Commands method*), 85  
[isCommand\(\)](#) (*supybot.callbacks.Plugin method*), 83  
[isCommandMethod\(\)](#) (*supybot.callbacks.Commands method*), 85  
[isCommandMethod\(\)](#) (*supybot.callbacks.Plugin method*), 83  
[isCtcp\(\)](#) (*in module supybot.ircmsgs*), 93  
[isDisabled\(\)](#) (*supybot.callbacks.Commands method*), 86  
[isDisabled\(\)](#) (*supybot.callbacks.Plugin method*), 83  
[isHalfop\(\)](#) (*supybot.irclib.ChannelState method*), 100  
[isHalfopPlus\(\)](#) (*supybot.irclib.ChannelState method*), 100  
[isNick\(\)](#) (*supybot.irclib.Irc method*), 96  
[isOp\(\)](#) (*supybot.irclib.ChannelState method*), 100  
[isOpPlus\(\)](#) (*supybot.irclib.ChannelState method*), 100  
[isVoice\(\)](#) (*supybot.irclib.ChannelState method*), 100  
[isVoicePlus\(\)](#) (*supybot.irclib.ChannelState method*), 100

## J

[join\(\)](#) (*in module supybot.ircmsgs*), 93  
[joiner\(\)](#) (*supybot.registry.CommaSeparatedListOfStrings method*), 103  
[joiner\(\)](#) (*supybot.registry.CommaSeparatedSetOfStrings method*), 104



- [joiner\(\)](#) (*supybot.registry.SeparatedListOf* method), [111](#)  
[joiner\(\)](#) (*supybot.registry.SpaceSeparatedListOf* method), [113](#)  
[joiner\(\)](#) (*supybot.registry.SpaceSeparatedListOfStrings* method), [114](#)  
[joiner\(\)](#) (*supybot.registry.SpaceSeparatedSetOfStrings* method), [114](#)  
[joins\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[Json](#) (class in *supybot.registry*), [106](#)
- ## K
- [key\(\)](#) (*supybot.callbacks.CanonicalNameDict* method), [85](#)  
[kick\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[kicks\(\)](#) (in module *supybot.ircmsgs*), [93](#)
- ## L
- [List](#) (*supybot.callbacks.Disabled* attribute), [86](#)  
[List](#) (*supybot.registry.CommaSeparatedListOfStrings* attribute), [103](#)  
[List](#) (*supybot.registry.CommaSeparatedSetOfStrings* attribute), [103](#)  
[List](#) (*supybot.registry.SeparatedListOf* attribute), [110](#)  
[List](#) (*supybot.registry.SpaceSeparatedListOf* attribute), [112](#)  
[List](#) (*supybot.registry.SpaceSeparatedListOfStrings* attribute), [113](#)  
[List](#) (*supybot.registry.SpaceSeparatedSetOfStrings* attribute), [114](#)  
[listCommands\(\)](#) (*supybot.callbacks.Commands* method), [86](#)  
[listCommands\(\)](#) (*supybot.callbacks.Plugin* method), [83](#)
- ## M
- [MalformedIrcMsg](#), [92](#)  
[many](#) (class in *supybot.commands*), [89](#)  
[messages](#) (*supybot.irclib.Batch* attribute), [101](#)  
[MetaSynchronizedAndFirewalled](#) (class in *supybot.callbacks*), [86](#)  
[modes](#) (*supybot.irclib.ChannelState* attribute), [100](#)  
[modes\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[monitor\(\)](#) (*supybot.irclib.Irc* method), [96](#)  
[monitoring](#) (*supybot.irclib.Irc* attribute), [95](#)
- ## N
- [name](#) (*supybot.irclib.Batch* attribute), [101](#)  
[name\(\)](#) (*supybot.callbacks.Commands* method), [86](#)  
[name\(\)](#) (*supybot.callbacks.Plugin* method), [83](#)  
[name\(\)](#) (*supybot.irclib.IrcCallback* method), [101](#)  
[NestedCommandsIrcProxy](#) (class in *supybot.callbacks*), [86](#)  
[network](#), [123](#)  
[network](#) (*supybot.irclib.Irc* attribute), [94](#)  
[network-specific](#), [124](#)  
[nick](#) (*supybot.ircmsgs.IrcMsg* attribute), [91](#)  
[nick\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[nicksToHostmasks](#) (*supybot.irclib.IrcState* attribute), [98](#)  
[nickToHostmask\(\)](#) (*supybot.irclib.IrcState* method), [98](#)  
[noIgnore](#) (*supybot.callbacks.PluginMixin* attribute), [87](#)  
[NonExistentRegistryEntry](#), [106](#)  
[NonNegativeInteger](#) (class in *supybot.registry*), [106](#)  
[noReply\(\)](#) (*supybot.callbacks.NestedCommandsIrcProxy* method), [87](#)  
[noReply\(\)](#) (*supybot.callbacks.RichReplyMethods* method), [89](#)  
[normalize\(\)](#) (*supybot.callbacks.CanonicalNameSet* method), [85](#)  
[normalize\(\)](#) (*supybot.callbacks.CanonicalString* method), [85](#)  
[NormalizedString](#) (class in *supybot.registry*), [107](#)  
[notice\(\)](#) (in module *supybot.ircmsgs*), [93](#)
- ## O
- [on\\_cap\\_end\(\)](#) (*supybot.irclib.IrcStateFsm* method), [99](#)  
[on\\_end\\_motd\(\)](#) (*supybot.irclib.IrcStateFsm* method), [99](#)  
[on\\_init\\_messages\\_sent\(\)](#) (*supybot.irclib.IrcStateFsm* method), [99](#)  
[on\\_sasl\\_auth\\_finished\(\)](#) (*supybot.irclib.IrcStateFsm* method), [99](#)  
[on\\_sasl\\_cap\(\)](#) (*supybot.irclib.IrcStateFsm* method), [99](#)  
[on\\_start\\_motd\(\)](#) (*supybot.irclib.IrcStateFsm* method), [99](#)  
[OnlySomeStrings](#) (class in *supybot.registry*), [108](#)  
[op\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[open\\_registry\(\)](#) (in module *supybot.registry*), [118](#)  
[ops](#) (*supybot.irclib.ChannelState* attribute), [100](#)  
[ops\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[optional](#) (class in *supybot.commands*), [89](#)  
[outFilter](#), [123](#)  
[outfilter](#), [123](#)  
[outFilter\(\)](#) (*supybot.callbacks.Plugin* method), [83](#)  
[outFilter\(\)](#) (*supybot.irclib.IrcCallback* method), [101](#)
- ## P
- [parent\\_batch](#) (*supybot.irclib.Batch* attribute), [101](#)  
[part\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[parts\(\)](#) (in module *supybot.ircmsgs*), [93](#)  
[password\(\)](#) (in module *supybot.ircmsgs*), [93](#)

ping() (in module supybot.ircmsgs), 93  
 plugin, 123  
 Plugin (class in supybot.callbacks), 83  
 PluginMixin (class in supybot.callbacks), 87  
 PluginRegexp (class in supybot.callbacks), 84  
 pong() (in module supybot.ircmsgs), 93  
 PositiveFloat (class in supybot.registry), 108  
 PositiveInteger (class in supybot.registry), 109  
 postTransition() (supybot.callbacks.Plugin method), 84  
 postTransition() (supybot.irclib.IrcCallback method), 101  
 pre\_command\_callbacks (supybot.callbacks.Commands attribute), 86  
 prefix (supybot.ircmsgs.IrcMsg attribute), 91  
 prettyPrint() (in module supybot.ircmsgs), 93  
 Privmsg (in module supybot.callbacks), 88  
 privmsg() (in module supybot.ircmsgs), 93  
 PrivmsgCommandAndRegexp (in module supybot.callbacks), 88  
 Probability (class in supybot.registry), 109  
 process() (in module supybot.commands), 90  
 Proxy (supybot.callbacks.Plugin attribute), 83  
 Proxy (supybot.callbacks.PluginMixin attribute), 87  
 public (supybot.callbacks.PluginMixin attribute), 87  
 Python Enhancement Proposals  
   PEP 8, 47

## Q

queue (supybot.irclib.Irc attribute), 95  
 queueBatch() (supybot.irclib.Irc method), 96  
 queueMsg() (supybot.irclib.Irc method), 96  
 queueMultilineBatches() (supybot.callbacks.ReplyIrcProxy method), 88  
 quit() (in module supybot.ircmsgs), 94

## R

Regexp (class in supybot.registry), 110  
 regexp\_wrapper() (in module supybot.commands), 90  
 regexps (supybot.callbacks.PluginRegexp attribute), 84  
 RegistryException, 110  
 registryValue() (supybot.callbacks.Plugin method), 84  
 registryValue() (supybot.callbacks.PluginMixin method), 87  
 remove() (supybot.callbacks.DisabledCommands method), 86  
 removeCallback() (supybot.irclib.Irc method), 96  
 removeCallback() (supybot.registry.Boolean method), 102

removeCallback() (supybot.registry.CommaSeparatedListOfStrings method), 103  
 removeCallback() (supybot.registry.CommaSeparatedSetOfStrings method), 104  
 removeCallback() (supybot.registry.Float method), 105  
 removeCallback() (supybot.registry.Integer method), 105  
 removeCallback() (supybot.registry.Json method), 106  
 removeCallback() (supybot.registry.NonNegativeInteger method), 107  
 removeCallback() (supybot.registry.NormalizedString method), 107  
 removeCallback() (supybot.registry.OnlySomeStrings method), 108  
 removeCallback() (supybot.registry.PositiveFloat method), 108  
 removeCallback() (supybot.registry.PositiveInteger method), 109  
 removeCallback() (supybot.registry.Probability method), 110  
 removeCallback() (supybot.registry.Regexp method), 110  
 removeCallback() (supybot.registry.SeparatedListOf method), 111  
 removeCallback() (supybot.registry.SeparatedListOf.Value method), 111  
 removeCallback() (supybot.registry.SpaceSeparatedListOf method), 113  
 removeCallback() (supybot.registry.SpaceSeparatedListOf.Value method), 112  
 removeCallback() (supybot.registry.SpaceSeparatedListOfStrings method), 114  
 removeCallback() (supybot.registry.SpaceSeparatedSetOfStrings method), 114  
 removeCallback() (supybot.registry.String method), 115  
 removeCallback() (supybot.registry.StringSurroundedBySpaces method), 116  
 removeCallback() (supybot.registry.StringWithSpaceOnRight method), 116  
 removeCallback() (supybot.registry.TemplatedString method), 117

[removeCallback\(\)](#) (*supybot.registry.Value method*), [117](#)  
[removeUser\(\)](#) (*supybot.irclib.ChannelState method*), [100](#)  
[replaceUser\(\)](#) (*supybot.irclib.ChannelState method*), [100](#)  
[replies\(\)](#) (*supybot.callbacks.NestedCommandsIrcProxy method*), [87](#)  
[replies\(\)](#) (*supybot.callbacks.RichReplyMethods method*), [89](#)  
[reply\(\)](#) (*in module supybot.callbacks*), [89](#)  
[reply\(\)](#) (*supybot.callbacks.NestedCommandsIrcProxy method*), [87](#)  
[reply\(\)](#) (*supybot.callbacks.ReplyIrcProxy method*), [88](#)  
[reply\\_env](#) (*supybot.ircmsgs.IrcMsg attribute*), [92](#)  
[replyError\(\)](#) (*supybot.callbacks.RichReplyMethods method*), [89](#)  
[ReplyIrcProxy](#) (*class in supybot.callbacks*), [88](#)  
[replySuccess\(\)](#) (*supybot.callbacks.RichReplyMethods method*), [89](#)  
[REQUEST\\_CAPABILITIES](#) (*supybot.irclib.Irc attribute*), [95](#)  
[REQUEST\\_EXPERIMENTAL\\_CAPABILITIES](#) (*supybot.irclib.Irc attribute*), [95](#)  
[requestCapabilities\(\)](#) (*supybot.irclib.Irc method*), [96](#)  
[reset\(\)](#) (*supybot.callbacks.Plugin method*), [84](#)  
[reset\(\)](#) (*supybot.irclib.Irc method*), [96](#)  
[reset\(\)](#) (*supybot.irclib.IrcCallback method*), [101](#)  
[reset\(\)](#) (*supybot.irclib.IrcMsgQueue method*), [102](#)  
[reset\(\)](#) (*supybot.irclib.IrcState method*), [98](#)  
[rest](#) (*class in supybot.commands*), [90](#)  
[reverse](#) (*class in supybot.commands*), [90](#)  
[RichReplyMethods](#) (*class in supybot.callbacks*), [88](#)  
[run\(\)](#) (*supybot.callbacks.CommandProcess method*), [85](#)  
[run\(\)](#) (*supybot.callbacks.CommandThread method*), [85](#)

## S

[sendMessage\(\)](#) (*supybot.irclib.Irc method*), [96](#)  
[SeparatedListOf](#) (*class in supybot.registry*), [110](#)  
[SeparatedListOf.Value](#) (*class in supybot.registry*), [111](#)  
[separators](#) (*supybot.callbacks.Tokenizer attribute*), [89](#)  
[server](#), [123](#)  
[server\\_tags](#) (*supybot.ircmsgs.IrcMsg attribute*), [92](#)  
[set\(\)](#) (*supybot.registry.Boolean method*), [102](#)  
[set\(\)](#) (*supybot.registry.CommaSeparatedListOfStrings method*), [103](#)  
[set\(\)](#) (*supybot.registry.CommaSeparatedSetOfStrings method*), [104](#)  
[set\(\)](#) (*supybot.registry.Float method*), [105](#)  
[set\(\)](#) (*supybot.registry.Integer method*), [105](#)  
[set\(\)](#) (*supybot.registry.Json method*), [106](#)  
[set\(\)](#) (*supybot.registry.NonNegativeInteger method*), [107](#)  
[set\(\)](#) (*supybot.registry.NormalizedString method*), [107](#)  
[set\(\)](#) (*supybot.registry.OnlySomeStrings method*), [108](#)  
[set\(\)](#) (*supybot.registry.PositiveFloat method*), [108](#)  
[set\(\)](#) (*supybot.registry.PositiveInteger method*), [109](#)  
[set\(\)](#) (*supybot.registry.Probability method*), [110](#)  
[set\(\)](#) (*supybot.registry.Regexp method*), [110](#)  
[set\(\)](#) (*supybot.registry.SeparatedListOf method*), [112](#)  
[set\(\)](#) (*supybot.registry.SeparatedListOf.Value method*), [111](#)  
[set\(\)](#) (*supybot.registry.SpaceSeparatedListOf method*), [113](#)  
[set\(\)](#) (*supybot.registry.SpaceSeparatedListOf.Value method*), [112](#)  
[set\(\)](#) (*supybot.registry.SpaceSeparatedListOfStrings method*), [114](#)  
[set\(\)](#) (*supybot.registry.SpaceSeparatedSetOfStrings method*), [115](#)  
[set\(\)](#) (*supybot.registry.String method*), [115](#)  
[set\(\)](#) (*supybot.registry.StringSurroundedBySpaces method*), [116](#)  
[set\(\)](#) (*supybot.registry.StringWithSpaceOnRight method*), [116](#)  
[set\(\)](#) (*supybot.registry.TemplatedString method*), [117](#)  
[set\(\)](#) (*supybot.registry.Value method*), [117](#)  
[setRegistryValue\(\)](#) (*supybot.callbacks.Plugin method*), [84](#)  
[setRegistryValue\(\)](#) (*supybot.callbacks.PluginMixin method*), [87](#)  
[setUserValue\(\)](#) (*supybot.callbacks.PluginMixin method*), [87](#)  
[setValue\(\)](#) (*supybot.registry.Boolean method*), [102](#)  
[setValue\(\)](#) (*supybot.registry.CommaSeparatedListOfStrings method*), [103](#)  
[setValue\(\)](#) (*supybot.registry.CommaSeparatedSetOfStrings method*), [104](#)  
[setValue\(\)](#) (*supybot.registry.Float method*), [105](#)  
[setValue\(\)](#) (*supybot.registry.Integer method*), [105](#)  
[setValue\(\)](#) (*supybot.registry.Json method*), [106](#)  
[setValue\(\)](#) (*supybot.registry.NonNegativeInteger method*), [107](#)  
[setValue\(\)](#) (*supybot.registry.NormalizedString method*), [107](#)  
[setValue\(\)](#) (*supybot.registry.OnlySomeStrings method*), [108](#)  
[setValue\(\)](#) (*supybot.registry.PositiveFloat method*), [109](#)  
[setValue\(\)](#) (*supybot.registry.PositiveInteger method*), [109](#)  
[setValue\(\)](#) (*supybot.registry.Probability method*), [110](#)

- `setValue()` (*supybot.registry.Regexp* method), 110
  - `setValue()` (*supybot.registry.SeparatedListOf* method), 112
  - `setValue()` (*supybot.registry.SeparatedListOf.Value* method), 111
  - `setValue()` (*supybot.registry.SpaceSeparatedListOf* method), 113
  - `setValue()` (*supybot.registry.SpaceSeparatedListOf.Value* method), 112
  - `setValue()` (*supybot.registry.SpaceSeparatedListOfStrings* method), 114
  - `setValue()` (*supybot.registry.SpaceSeparatedSetOfStrings* method), 115
  - `setValue()` (*supybot.registry.String* method), 115
  - `setValue()` (*supybot.registry.StringSurroundedBySpaces* method), 116
  - `setValue()` (*supybot.registry.StringWithSpaceOnRight* method), 116
  - `setValue()` (*supybot.registry.TemplatedString* method), 117
  - `setValue()` (*supybot.registry.Value* method), 117
  - `SilentError`, 89
  - `SimpleProxy` (in module *supybot.callbacks*), 89
  - `sorted` (*supybot.callbacks.Disabled* attribute), 86
  - `SpaceSeparatedListOf` (class in *supybot.registry*), 112
  - `SpaceSeparatedListOf.Value` (class in *supybot.registry*), 112
  - `SpaceSeparatedListOfStrings` (class in *supybot.registry*), 113
  - `SpaceSeparatedSetOfStrings` (class in *supybot.registry*), 114
  - `Spec` (class in *supybot.commands*), 90
  - `specific`, 123
  - `split_args()` (in module *supybot.ircmsgs*), 94
  - `splitter()` (*supybot.registry.CommaSeparatedListOfStrings* method), 103
  - `splitter()` (*supybot.registry.CommaSeparatedSetOfStrings* method), 104
  - `splitter()` (*supybot.registry.SeparatedListOf* method), 112
  - `splitter()` (*supybot.registry.SpaceSeparatedListOf* method), 113
  - `splitter()` (*supybot.registry.SpaceSeparatedListOfStrings* method), 114
  - `splitter()` (*supybot.registry.SpaceSeparatedSetOfStrings* method), 115
  - `startedAt` (*supybot.irclib.Irc* attribute), 94
  - `startedSync` (*supybot.irclib.Irc* attribute), 95
  - `state` (*supybot.irclib.Irc* attribute), 95
  - `String` (class in *supybot.registry*), 115
  - `StringSurroundedBySpaces` (class in *supybot.registry*), 115
  - `StringWithSpaceOnRight` (class in *supybot.registry*), 116
  - `supported` (*supybot.irclib.IrcState* attribute), 97
  - `supybot.callbacks` (module), 84
  - `supybot.commands` (module), 89
  - `supybot.irclib` (module), 101
  - `supybot.ircmsgs` (module), 91
  - `supybot.registry` (module), 102
- ## T
- `tag()` (*supybot.ircmsgs.IrcMsg* method), 92
  - `tagged()` (*supybot.ircmsgs.IrcMsg* method), 92
  - `tags` (*supybot.ircmsgs.IrcMsg* attribute), 92
  - `takeMsg()` (*supybot.irclib.Irc* method), 97
  - `TemplatedString` (class in *supybot.registry*), 116
  - `thread()` (in module *supybot.commands*), 90
  - `threaded` (*supybot.callbacks.PluginMixin* attribute), 88
  - `time` (*supybot.ircmsgs.IrcMsg* attribute), 91
  - `tokenize()` (in module *supybot.callbacks*), 84
  - `Tokenizer` (class in *supybot.callbacks*), 89
  - `topic` (*supybot.irclib.ChannelState* attribute), 99
  - `topic()` (in module *supybot.ircmsgs*), 94
  - `type` (*supybot.irclib.Batch* attribute), 101
- ## U
- `unAction()` (in module *supybot.ircmsgs*), 94
  - `unaddressedRegexps` (*supybot.callbacks.PluginRegexp* attribute), 84
  - `unban()` (in module *supybot.ircmsgs*), 94
  - `unbans()` (in module *supybot.ircmsgs*), 94
  - `UNINITIALIZED` (*supybot.irclib.IrcStateFsm.States* attribute), 99
  - `unmonitor()` (*supybot.irclib.Irc* method), 97
  - `urlSnarfer()` (in module *supybot.commands*), 90
  - `user` (*supybot.ircmsgs.IrcMsg* attribute), 91
  - `user()` (in module *supybot.ircmsgs*), 94
  - `users` (*supybot.irclib.ChannelState* attribute), 100
  - `userValue()` (*supybot.callbacks.PluginMixin* method), 88
- ## V
- `Value` (class in *supybot.registry*), 117
  - `Value` (*supybot.callbacks.Disabled* attribute), 86
  - `Value` (*supybot.registry.CommaSeparatedListOfStrings* attribute), 103
  - `Value` (*supybot.registry.CommaSeparatedSetOfStrings* attribute), 103
  - `Value` (*supybot.registry.SpaceSeparatedListOfStrings* attribute), 113
  - `Value` (*supybot.registry.SpaceSeparatedSetOfStrings* attribute), 114
  - `voice()` (in module *supybot.ircmsgs*), 94
  - `voices` (*supybot.irclib.ChannelState* attribute), 100
  - `voices()` (in module *supybot.ircmsgs*), 94

## W

[who\(\)](#) (in module *supybot.ircmsgs*), 94  
[whois\(\)](#) (in module *supybot.ircmsgs*), 94  
[whowas\(\)](#) (in module *supybot.ircmsgs*), 94  
[with\\_traceback\(\)](#) (*supybot.registry.InvalidRegistryFile* method), 105  
[with\\_traceback\(\)](#) (*supybot.registry.InvalidRegistryName* method), 106  
[with\\_traceback\(\)](#) (*supybot.registry.InvalidRegistryValue* method), 106  
[with\\_traceback\(\)](#) (*supybot.registry.NonExistentRegistryEntry* method), 106  
[with\\_traceback\(\)](#) (*supybot.registry.RegistryException* method), 110  
[wrap\(\)](#) (in module *supybot.commands*), 90

## Z

[zombie](#) (*supybot.irclib.Irc* attribute), 94