PyPsd Documentation

Release 3.0

Rizon Chat Network

May 24, 2017

Contents

1 Using PyPsd **3**

1.1 Getting Started 3

1.2 Frequently Asked Questions 4

1.3 Support 5

2 Module-specific Info **7**

2.1 Internets: Weather/Forecast Commands 7

3 Developing PyPsd **9**

3.1 Low-Level Modules 9

3.2 User Modules 11

PyPsd provides clean and easy-to-use IRC services. It runs under PyPy, requiring a MySQL or MariaDB database and a TS6-based IRCd. The code's open source under the BSD 3-clause license, and available on BitBucket.

PyPsd was written for Rizon Chat Network, and started by Radicand in early 2009. It was initially created to provide DNS Blacklist services, but as time went on more modules and features were added, and it will hopefully become a nice all-round addition to any IRC network.

Warning: Due to how PyPsd is currently coded, you probably cannot integrate it into much except Plexus, without some level of modification (due to some specific TS6 implementation quirks). This may change in the future, and users wishing to put PyPsd up on their own networks are likely to have difficulties right now.

The auth system (used by most user-facing modules to join and part requested channels) is also highly linked to Rizon's services, with `CHANSERV WHY` built by/for Rizon and not available on stock Anope. However, this is likely to change in the future, as Anope 2.0 introduces `CHANSERV STATUS` which PyPsd is being built to support (this work is being done on the new-services-flag branch)

The documentation's organised into two sections below, the first for users of PyPsd and the second for developers looking to work on or write their own modules for PyPsd.

CHAPTER 1

Using PyPsd

Getting Started

PyPsd is fairly easy to work with, providing you have the right things in place.

Prerequisites

- Unix-like operating system (Linux supported, others may be spotty)
- PyPy/CPython 2.7, with PyMySQL and NetworkX
- MySQL or MariaDB database (MariaDB not officially supported, but confirmed to work)
- Git (installed on the command line, used for dynamic version generation)
- Plexus, other TS6 IRCd's not tested
- Anope 1.8 (with custom-built `CHANSERV WHY` command), soon to support stock Anope 2.0

Installation

Installing PyPsd is easy – first off, we install our dependencies. With PyPy and `easy_install` installed:

```
$ pypy -m easy_install pymysql

$ pypy -m easy_install networkx
```

Next, clone the PyPsd git repository:

```
$ git clone https://bitbucket.org/rizon/pypsd.git
Cloning into 'pypsd'...
remote: Counting objects: 163, done.
remote: Compressing objects: 100% (159/159), done.
remote: Total 163 (delta 0), reused 138 (delta 0)
```

```
Receiving objects: 100% (163/163), 237.13 KiB | 31.00 KiB/s, done.
Checking connectivity... done

$ cd pypsd
```

Copy the base config file, and modify it to fit your purposes:

```
$ cp config.ini.sample config.ini

$ vim config.ini
```

Frequently Asked Questions

I already run Anope/Atheme, is there a reason I should run another services package?

PyPsd isn't an ordinary services package. Instead of trying to provide NickServ, ChanServ and other services that almost every network already runs, PyPsd provides services that are usually overlooked, or provided using custom-built software. This includes, but isn't limited to:

- CTCP VERSION/WEBSITE statistics tracking, and banning based off VERSION responses
- Internets bot, to provide quick and simple Google, Youtube, Weather, and Translation services for your users
- Quotes bot, to provide a nice and simple utility for your users to add, track, and recall quotes from their channel

In particular, having bots that provide these sorts of services help dissuade users from bringing in their own bots to do the job for them. Isn't it nicer to have one single, unified experience across your network, and have one utility everyone can use instead of a bunch more bots connected to your network?

Warning: PyPsd's module auth system does not support Atheme by default. You will need to code the support in yourself, for the modules you wish to run. Our auth system also does not work with stock Anope 1.9 (since we use a custom command to check channel-user authorization), however, we will soon support stock Anope 2.0.

Isn't Python too slow to run these for any decently-sized network?

Surprisingly not!

While regular Python (CPython) may be too slow to provide decent network services, PyPsd can be run under an alternative interpreter called PyPy (This is in fact the recommended way to run PyPsd). This allows it to scale to quite large networks, proving itself by providing services for Rizon Chat Network which has an average of around 20k users.

If I write a module for my network, do I *need* to release it to everyone?

We would love it if you could, or if you would even try to contribute back to the primary PyPsd repository so that everyone running the software can get the great new features!

However, it is not a requirement. Because PyPsd is licensed under the BSD 3-clause license, you have the flexibility to keep any changes to yourself if you want to.

Support

Getting Help

The easiest way to get help is through the `#dev` channel on Rizon. Most of the developers hang out there, and are responsive to any questions or queries that are brought up. Note, however, that if your question is answered either here, or with a fairly simple web search, we are likely to be annoyed.

If you wish to report a bug, it may be better off to create a new issue on Bitbucket and yell at us to fix it ;-)

CHAPTER 2

Module-specific Info

Internets: Weather/Forecast Commands

The Internets Weather/Forecast commands use OpenWeatherMap for their weather data. In addition, weather and forecast support the ability to accept U.S. ZIP codes as input.

Accepting U.S. ZIP Codes

In PyPsd's main directory, there's a `weather-zipcodes.py` script. This will automagically connect to the database in your PyPsd config file, and import all of our ZIP codes (about 43k in total).

Script options:

- `--drop`: Drop the ZIP codes currently in the database
- `--debug`: Print each ZIP code that we import, mostly used for debugging

Example script output

Importing table:

```
$ ./populate-zipcodes.py
ZIP Table does not yet exist.
ZIP Table created.
Populating ZIP Table
1000 rows inserted
2000 rows inserted
3000 rows inserted
4000 rows inserted
...
41000 rows inserted
42000 rows inserted
```

```
43000 rows inserted
43204 rows inserted in total
```

Dropping table:

```
$ ./populate-zipcodes.py --drop
Current ZIP table dropped.
```

Debug information:

```
$ ./populate-zipcodes.py --debug
ZIP Table does not yet exist.
ZIP Table created.
Populating ZIP Table
Inserted ZIP code 00210 : -71.013202 : 43.005895
Inserted ZIP code 00211 : -71.013202 : 43.005895
Inserted ZIP code 00212 : -71.013202 : 43.005895
Inserted ZIP code 00213 : -71.013202 : 43.005895

Debug numbers represent:    ZIP Code : Latitude : Longitude
```

License

The ZIP code data itself is in `data/weather-zipcodes.csv`. This data is from the CivicSpace US ZIP Code Database, and licensed under the Creative Commons Attribution-ShareAlike license as specified in the readme file, `data/weather-zipcodes.README`.

CHAPTER 3

Developing PyPsd

Low-Level Modules

This is the general module API – the lower-level stuff, the stuff that UModule interfaces with.

Naming

Module filenames must be in the format of `psm_yourmodulenamehere.py` preferrably all lowercase.

Module classes must be in the format of `PSModule_yourmodulenamehere`, with both yourmodulenamehere's being the same case.

For example, I may name my example module `psm_example.py`, with the class name `PSModule_example`.

Imports

All modules must import:

```
from psmodule import *
```

Class

All modules must extend `PSModule`

Class Variables

You should implement class variables:

- `VERSION` = (number), up to you.

Additionally, the default `__init__` sets several useful class variables for you to use in your module:

- **parent = object (received from __init__, your parent server** object, e.g., TS6Protocol()).
- **config = object (received from __init__, the same** configuration object used in the parent server object).
- **logchan = string (is set in __init__, channel from config** where your bot users reside if you use one).
- **log = object (is set in __init__, file/stdout logger (usage:** log.error(msg)|log.info(msg). . . etc, see python logging lib for details).
- **dbp = object (is set in __init__, is the parent's database** pointer) – NB you should always make your own database if your module needs one.

Commands

There is only one sort of 'command' in the general module API. In UModule, these are called admin commands (acmds), and we will refer to them as such here.

acmd

Admin Commands are actual pypsd commands. To create an acmd_something.py library, simply add the functions you wish, and return the (function, usagestring) as a tuple from `getCommands()`

ACL flags, and Permissions

Each acmd has 'ACL flags' that you need to specify. This is essentially how the permissions in PyPsd currently work:

We can have 52 ACL flags, from a-z and A-Z.

When someone creates a module, they choose a character `(a-zA-Z)` that nobody else has picked yet, and that is essentially their module's 'access control' mode character. For instance, for my new Twitter module, I might pick `'r'`.

After you pick your character, make sure your commands have it in the 'permission' entry, as such:

```
def getCommands(self):
    return (
        ('disable', {
        'permission' : 'r',
        'callback' : self.cmd_disable,
        'usage' : '- Disable Twitter functionality'}),
    )
```

This means that users will need to have `'r'` in their ACL flags list to run the `twitter.disable` acmd.

Examples of ACL flags lists:

- IAmAGod: `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`
- EverythingButShutdown: `bcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`
- Power: `efinopstuwxyzABCDEFGHIJKLMQRSTUVW`
- KindaOkay: `ioqswyzCDEKQSTU`
- MidnightShiftGuy: `ghvwDFRTVZ`
- TheTwitterGuy: `r`

It may also be useful to give different commands in the same module different flags. For instance, you may want to let someone restart your module, but not shut it down completely.

ACL flags list

Here's a list of what different flags are used to control:

- `r`: Core - Shutting down, and reloading PyPsd itself
- `m`: Core – Module loading, unloading, and reloading
- `l`: Core – Channel and user lookups
- `a`: ACL – Removing and adding ACL flags from users
- `b`: Bouncer – Blacklists based on CIDR blacklists
- `b`: ProxyBridge – Scan users for proxies, and take action
- `d`: Debugger – Debugging PyPsd, log searching and terminal commands
- `d`: DNS Blacklist – Controlling a DNS blacklist
- `e`: eRepublik – Online, global strategy game
- `e`: e-Sim – Another online game, like eRepublik
- `e`: Internets – Controls access to all of Interents' admin commands
- `j`: LimitServ – Controls number of users in channels?
- `j`: Ninja – Keeps privmsgs from being posted heaps of times
- `j`: Quotes – Keeping a local quote database
- `j`: Trivia – Running trivia games
- `r`: DynLogLevel – Changing PyPsd's logging level
- `r`: WALLOPS – Redirects WALLOPS to a channel
- `s`: NetStats – Keeps network statistics
- `v`: CTCP – CTCP info collecting and blacklisting
- `w`: WikiMonitor – Monitors Wikis
- `x`: Xray – Scans nicks, channel names, etc given provided regexes
- `A`: AKILL – AKILLing users, removing AKILLs
- `B`: ListBots – Lists important bots for users
- `D`: Deaf – TODO
- `M`: HTTP Monitor – Monitors HTTP
- `N`: NetAdmin – Sends RAW IRC messages. Be very afraid
- `N`: ProxyBridge – Reloading Proxybridge's config information
- `N`: UIDfix – Fixing modules' UIDs, low-level scary scary

User Modules

UModule is a new framework, created to unify the multiple copies of frameworks for every module. It is currently in development, on the `umodule` branch, and not yet in trunk.

Naming

Module filenames must be in the format of `psm_yourmodulenamehere.py` preferrably all lowercase.

Module classes must be in the format of `PSModule_yourmodulenamehere`, with both yourmodulenamehere's being the same case.

For example, I may name my example module `psm_example.py`, with the class name `PSModule_example`.

Imports

All modules must import:

```
from umodule import UModule
```

Class

All modules must extend `UModule`

Class Variables

You should implement class variables:

- `NAME` = (string), no whitespace. Will be automagically set to the string after `'PSModule_'` in the class name if not otherwise set, so you shouldn't need to change it. (eg: `PSModule_twitter` would result in a NAME of `'twitter'`). Must be the same as your module's folder name.
- `DISPLAY_NAME` = (string), name of this module when shown to the user. Used, for instance, in info lists and such. Defaults to `NAME.title()`, but examples of a few 'special' ones may include `eRepublik` or `e-Sim`.
- `VERSION` = (number), up to you.
- `DEVELOPERS` = (string), `'Dev Elepor <dev@elep.or>, Some Onelse <som@wan.wan>'`.

You may choose to implement these class variables:

- `parser` = (optparse.OptionParser). Used for custom option settings when commands are parsed.
- `parser_option` = (optparse.Option). See above. Look through `/esim/esimparser.py` for examples.
- `LISTBOTS` = (list). If specified, `/psm_listbots.py` will display this name/description in its list of network bots.

Additionally, the default `__init__` sets several useful class variables for you to use in your module:

- `parent` = object (received from `__init__`, your parent server object, e.g., TS6Protocol()).
- `config` = object (received from `__init__`, the same configuration object used in the parent server object).
- `logchan` = string (is set in `__init__`, channel from config where your bot users reside if you use one).
- `log` = object (is set in `__init__`, file/stdout logger (usage: log.error(msg)|log.info(msg)...etc, see python logging lib for details).
- `dbp` = object (is set in `__init__`, is the parent's database pointer) – NB you should always make your own database if your module needs one.

config.ini

UModule assumes that users want to have a virtual user connect to the server.

The following info must be present in config.ini for this to work, where NAME is the same as the class variable NAME, above.

```
[NAME]
nick: twitter
user: twitter
host: 140.or.bust
gecos: Network Services Bot
modes: +SUoipqNx
nspass: twit
channel: #a
```

Libraries

A few different convenience libraries are loaded by default. To find out how to use these, simply look at how other umodule-based modules implement them, and at the code in `libs/sys_<library>.py` itself. These are implemented as class variables, as follows:

ones you are likely to call yourself

- `auth`: Handles situations where you need to verify the user is the founder of a channel.
- `channels`: Handles channels your fake user is 'allowed' in, and has been requested in.
- `log`: Provides simple, consistent logging, such as self.log.debug('string').
- `options`: Handles database-stored module options, with a simple interface.

background libs

- `antiflood`: Makes sure users don't flood you to death with messages.
- `users`: Keeps track of banned users.

Commands

There are two sorts of commands in UModules, Admin Commands (acmd), and User Commands (ucmd). This section talks about how to properly use the two types in a UModule.

acmd

Admin Commands, as they're called, are actual pypsd commands. To create an acmd_something.py library, simply add the functions you wish, and return the (function, usagestring) as a tuple from `get_commands()`, like this:

```python
def admin_stats(self, source, target, pieces):
    self.msg(target, 'Registered users: @b%d@b.' % len(self.users.list_all()))
    self.msg(target, 'Registered channels: @b%d@b.' % len(self.channels.list_all()))
    return True
```

```
def get_commands():
    return {
        'stats': (admin_stats, 'counts registered users and channels')
    }
```

This file, acmd_something, is put in your umodule folder. Then, in your module's **init** block, you use the `load_acmd()` function to load your library:

```
from umodule import UModule
from libs import acmd_shared

class PSModule_twitter(UModule):
    def __init__(self, parent, config):
        UModule.__init__(self, parent, config)
        self.load_acmd('something', acmd_something)
```

After that, UModule will take care of the rest, including setting each command to only be runnable by pypsd admins only. You can additionally set custom pypsd permissions and such per command by adding a dict the the end of a command's `get_commands()` tuple. Like this:

```
def get_commands():
    return {
        'stats': (admin_stats, 'counts registered users and channels', {'permission':
↪'e'})
    }
```

If you do require special acmd loading, you may load your own custom commands using UModule's `register_command()` manually. Look into the `register_command()` docstring yourself for specific information on how to use it

ucmd

User Commands are commands that are entirely handled by UModule itself. These are what all your users will be calling, and how your users will interact with your module. There are a number of commands that are shared between various modules, such as help, info, request, and remove. Let's take a look at how those work:

```
from ucmd_manager import *

def shared_info(self, manager, zone, opts, arg, channel, sender, userinfo):
    message = '@sep @bRizon %s Bot@b @sep @bVersion@b %s @sep @bDevelopers@b %s @sep'
↪% (self.NAME, self.VERSION, self.DEVELOPERS)
    self.notice(sender, message)

def shared_help(self, manager, zone, opts, arg, channel, sender, userinfo):
    command = arg.lower()
    for line in self.get_help(zone, command):
        self.notice(sender, line)

class SharedCommandManager(CommandManager):
    command_list = {
        # info's key here, is a string. This is because the only name for this␣
↪command is 'info'.
        'info': (shared_info, CMD_ALL, ARG_NO|ARG_OFFLINE, 'Displays version and␣
↪author information', []),

        # help's key, however, is a tuple containing 'help' and 'hello'. The primary␣
↪name of this command is 'help', but this shows that 'hello' will also trigger this␣
↪command. Which name comes first will be the primary one displayed in help output.
```

```
        ('help', 'hello'): (shared_help, CMD_ALL,  ARG_OPT|ARG_OFFLINE, 'Displays␣
↪available commands and their usage', []),
    }
```

Each function that receives UCMDs has a number of arguments, namely: * `self`: The UModule itself * `manager`: The Command Manager the command is a part of * `zone`: The ucmd zone the command came from (`CMD_PUBLIC`, or `CMD_PRIVATE`) * `opts`: Option dictionary * `arg`: Arguments * `channel`: Command's target. Who the user was sending the message to in the first place * `sender`: Nick of the user who sent the message * `userinfo`: Sending user's userinfo

In a similar sort of fashion, a `command_list` value is a list containing these values: * `handler`: Handler function. Receives all those arguments above * `zone`: Zone the command can work in * `args`: Argument settings * `description`: Description, used for help messages * UNKNOWN, look into this a bit later * `usage`: Not shown here, this can either be a string or a list, containing usage strings. eg: `['#channel', 'nick']` or just `'#channel/nick'`. The main difference here is that list items will be shown on totally new lines, and a single string will be shown on just a single line

Argument Types

Zones

You may have noticed the zone argument in a few of the ucmd methods. This is how we differentiate whether a command can be used publicly (in a channel), privately (privmsg right to module user), or both.

- `CMD_PUBLIC`: Shows the command can be used publicly
- `CMD_PRIVATE`: Shows the command can be used privately
- `CMD_ALL`: Both of the above zones OR'd together, meaning both public and private

Args

TODO: Description here

Subsystems

Subsystems are blocks of code that manage stuff within your UModule. `self.auth`, `self.elog`, and `self.channels` are examples of a few default subsystems. Subsystems have database access, as well as a few other specific options and functions, setup for them automagically.

If you want to disable a default subsystem, to use your own in place of it, or to just do something different, add the module's name to your class variable `disabled_default_modules`, as such:

```
class MyAwesomeModule(UModule):
    u_settings = {
        'disabled_default_modules': ['antiflood', 'users']
    }
```

Be aware, though, that the default subsystems are quite tightly integrated with each other. Unloading, for instance, the `options` subsystem is a *very* bad idea unless you're going to add your own `options` subsystem in place of it. (note that you would need to add the subsystem before calling `UModule.__init__`, and load it as a `core` subsystem, otherwise all the other modules would fail. This is what I mean when I say it's really bad to unload certain subsystems)

To bind your own manager to be loaded, simply call `bind_subsystem`. It will be automatically loaded, and accessable at the name you give. For instance, this:

```
self.bind_subsystem('cool', sys_coolguys.CoolManager)
```

would allow you to access your cool manager at `self.cool` once it was loaded by `self.startup`. This is why you check whether your module is initialized and online before doing stuff, because the managers you try to access may not even exist before then.

Subsystem Priorities

By default, there are three 'priorities' of subsystems: * `core`: Primarily, subsystems that are loaded first, and are required by other lower-level subsystems such as `elog` and `options`. * `normal`: By default, these contain the useful subsystems that modules use, such as `channels`, `users`, `auth`, and `antiflood`. This is the default priority for newly added subsystems. * `minor`: No subsystems are in this group by default. This is intended for things like module APIs, that may make use of the default `core` and/or `normal` subsystems.

An example of both loading subsystems with different priorities, and providing kwargs to subsystems on start, is below. Keep in mind that for regular positional args, you would simply add something like `args=[2, 4, 5]`.

```
self.bind_subsystem('options', sys_options.OptionManager, priority='core')

self.bind_subsystem('weather', sys_weather.Weather, priority='minor', kwargs={
    'key': self.config.get('internets', 'key_wunderground')
})
```

Event Hooks

General hooks

Modules can hook into the parent code at any point. Currently hooks exist for `uid` (when a user connects), `privmsg` (on receiving a message), `notice` (on receiving a notice), and a few other events.

You can add extra hook points by modifying pseudoserver.py (see irc_UID() for proper usage). Your hooks must be in the following format:

```
def yourmodulenamehere_hooknamehere(self, prefix, params)
```

Where hooknamehere is whatever you want to call your hook. `prefix` and `params` are what get passed from the source hook method. It is of utmost importance to prefix your hooks with your module's name, otherwise the unloading code will break.

After writing your hook method, you add it in a tuple of tuples to your getHooks() definition as in the example below:

```
def getHooks(self):
    return (('uid', self.bopm_SCANUSER),)
```

UModule-specific hooks

UModule has a few default hook functions that *must* be run when certain hooks happen. These, currently, are `umodule_TMODE`, `umodule_PRIVMSG`, and `umodule_NOTICE`. These can either be passed directly as the hook function, or simply called from the module-specific hook function, as this:

```
def example_NOTICE(self, prefix, params):
    self.umodule_NOTICE(prefix, params)
    pass  # do whatever else here
```

If not using just the default hooks, the getHooks function must be manually specified, including at least the hooks `tmode`, `privmsg`, and `notice`. An example of such a basic getHooks call is this:

```
def getHooks(self):
    return (('tmode', self.umodule_TMODE),  # default
            ('notice', self.example_NOTICE),  # custom hook
            ('privmsg', self.umodule_PRIVMSG))  # default
```

If you require a more complete example of how this works, look into `psm_example.py`.