irc

Release 15.1.2.dev0+gbcd35df.d20170419

Apr 19, 2017

Contents

CHAPTER 1

History

15.1.1

19 Apr 2017

- New `send_items` method takes star args for simplicity in the syntax and usage.

15.1

19 Apr 2017

- Introduce `ServerConnection.send_items`, consolidating common behavior across many methods previously calling `send_raw`.

15.0.6

20 Dec 2016

- Now publish documentation to Read The Docs.

15.0.5

15 Nov 2016

- #119: Handle broken pipe exception in IRCClient _send() (server.py).

15.0.4

28 Oct 2016

- #116: Correct invocation of execute_every.

15.0.3

07 Oct 2016

- #115: Fix AttributeError in `execute_at` in scheduling support.

15.0.2

28 Sep 2016

- #113: Use preferred scheduler in the bot implementation.

15.0.1

12 Sep 2016

- Deprecated calls to Connection.execute_* and Reactor.execute_*. Instead, call the equivalently-named methods on the reactor's scheduler.

15.0

12 Sep 2016

- The event scheduling functionality has been decoupled from the client.Reactor object. Now the reactor will construct a Scheduler from the scheduler_class property, which must be an instance of irc.schedule.IScheduler.

  The `_on_schedule` parameter is no longer accepted to the Reactor class. Implementations requiring a signal during scheduling should hook into the `add` method of the relevant scheduler class.

- Moved the underlying scheduler implementation to tempora, allowing it to be re-used for other purposes.

14.2.2

11 Jun 2016

- Issue #98: Add an ugly hack to force `build_sphinx` command to have the requisite libraries to build module documentation.

14.2.1

11 Jun 2016

- Issue #97: Restore `irc.buffer` module for compatibility.
- Issue #95: Update docs to remove missing or deprecated modules.
- Issue #96: Declare Gitter support as a badge in the docs.

14.2

16 Apr 2016

- Moved buffer module to jaraco.stream for use in other packages.

14.1

28 Feb 2016

- `SingleServerIRCBot` now accepts a `recon` parameter implementing a ReconnectStrategy. The new default strategy is ExponentialBackoff, implementing an exponential backoff with jitter. The `reconnection_interval` parameter is now deprecated but retained for compatibility. To customize the minimum time before reconnect, create a custom ExponentialBackoff instance or create another ReconnectStrategy object and pass that as the `recon` parameter. The `reconnection_interval` parameter will be removed in future versions.
- Issue #82: The `ExponentialBackoff` implementation now protects from multiple scheduled reconnects, avoiding the issue where reconnect attempts accumulate exponentially when the bot is immediately disconnected by the server.

14.0

15 Feb 2016

- Dropped deprecated constructor `connection.Factory.from_legacy_params`. Use the natural constructor instead.
- Issue #83: `connection.Factory` no longer attempts to bind before connect unless a bind address is specified.

13.3.1

31 Dec 2015

- Now remove mode for owners, halfops, and admins when the user is removed from a channel.
- Refactored the Channel class implementation for cleaner, less repetitive code.
- Expanded tests coverage for Channel class.

13.3

31 Dec 2015

- Issue #75: In `irc.bot`, add support for tracking admin status (mode 'a') in channels. Use `channel.is_admin` or `channel.admins` to identify admin users for a channel.
- Removed deprecated irc.logging module.

13.2

20 Nov 2015

- Moved hosting to github.

13.1.1

04 Oct 2015

- Issue #67: Fix infinite recursion for `irc.strings.IRCFoldedCase` and `irc.strings.lower`.

13.1

23 Aug 2015

- Issue #64: ISUPPORT PREFIX now retains the order of permissions for each prefix.

13.0

21 Aug 2015

- Updated `schedule` module to properly support timezone aware times and use them by default. Clients that rely on the timezone naïve datetimes may restore the old behavior by overriding the `schedule.now` and `schedule.from_timestamp` functions like so:

  schedule.from_timestamp = datetime.datetime.fromtimestamp schedule.now = datetime.datetime.now

  Clients that were previously patching `schedule.DelayedCommand.now` will need to instead patch the aforementioned module-global methods. The classmethod technique was a poor interface for effectively controlling timezone awareness, so was likely unused. Please file a ticket with the project for support with your client as needed.

12.4.2

01 Jul 2015

- Bump to jaraco.functools 1.5 to throttler failures in Python 2.

12.4

01 Jul 2015

- Moved `Throttler` class to jaraco.functools 1.4.

12.3

25 Jun 2015

- Pull Request #33: Fix apparent escaping issue with IRCv3 tags.

12.2

25 May 2015

- Pull Request #32: Add numeric for WHOX reply.
- Issue #62 and Pull Request #34: Add support for tags in message processing and `Event` class.

12.1.2

25 Apr 2015

- Issue #59: Fixed broken references to irc.client members.
- Issue #60: Fix broken initialization of `irc.server.IRCClient` on Python 2.

12.1.1

10 Mar 2015

- Issue #57: Better handling of Python 3 in testbot.py script.

12.1

28 Feb 2015

- Remove changelog from package metadata.

12.0

25 Feb 2015

- Remove dependency on jaraco.util. Instead depend on surgical packages.
- Deprecated `irc.logging` in favor of `jaraco.logging`.
- Dropped support for Python 3.2.

11.1.1

19 Feb 2015

- Issue #55: Correct import error on Python 2.7.

11.1

18 Feb 2015

- Decoding errors now log a warning giving a reference to the `Decoding Input` section of the readme.

11.0

15 Nov 2014

- Renamed `irc.client.Manifold` to `irc.client.Reactor`. Reactor better reflects the implementation as a reactor pattern <. This name makes it's function much more clear and inline with standard terminology.
- Removed deprecated `manifold` and `irclibobj` properties from Connection. Use `reactor` instead.
- Removed deprecated `ircobj` from `SimpleIRCClient`. Use `reactor` instead.

10.1

13 Nov 2014

- Added `ServerConnection.as_nick`, a context manager to set a nick for the duration of the context.

10.0

27 Oct 2014

- Dropped support for Python 2.6.
- Dropped `irc.client.LineBuffer` and `irc.client.DecodingBuffer` (available in `irc.client.buffer`).
- Renamed `irc.client.IRC` to `irc.client.Manifold` to provide a clearer name for that object. Clients supporting 8.6 and later can use the `Manifold` name. Latest clients must use the `Manifold` name.
- Renamed `irc.client.Connection.irclibobj` property to `manifold`. The property is still exposed as `irclibobj` for compatibility but will be removed in a future version.
- Removed unused `irc.client.mask_matches` function.
- Removed unused `irc.client.nick_characters`.
- Added extra numerics for 'whoisaccount' and 'cannotknock'.

9.0

22 Oct 2014

- Issue #46: The `whois` command now accepts a single string or iterable for the target.
- NickMask now returns `None` when user, host, or userhost are not present. Previously, an `IndexError` was raised. See Pull Request '#26 <https://bitbucket.org/jaraco/irc/pull-request/26>'_ for details.

8.9

26 Apr 2014

Documentation is now published at https://pythonhosted.org/irc.

8.8

25 Apr 2014

- Issue #35: Removed the mutex during process_once.
- Issue #37: Deprecated buffer.LineBuffer for Python 3.

8.7

24 Apr 2014

- Issue #34: Introduced `buffer.LenientDecodingLineBuffer` for handling input in a more lenient way, preferring UTF-8 but falling back to latin-1 if the content cannot be decoded as UTF-8. To enable it by default for your application, set it as the default decoder:

```
irc.client.ServerConnection.buffer_class = irc.buffer.LenientDecodingLineBuffer
```

8.6

21 Apr 2014

- Introduced 'Manifold' as an alias for irc.client.IRC. This better name will replace the IRC name in a future version.
- Introduced the 'manifold' property of SimpleIRCClient as an alias for ircobj.
- Added 'manifold_class' property to the client.SimpleIRCClient to allow consumers to provide a customized Manifold.

8.5.4

17 Nov 2013

- Issue #32: Add logging around large DCC messages to facilitate troubleshooting.

- Issue #31: Fix error in connection wrapper for SSL example.

8.5.3

22 Sep 2013

- Issue #28: Fix TypeError in version calculation in irc.bot CTCP version.

8.5.2

22 Sep 2013

- Updated DCC send and receive scripts (Issue #27).

8.5.1

16 Aug 2013

- Fix timestamp support in `schedule.DelayedCommand` construction.

8.5

10 Aug 2013

- `irc.client.NickMask` is now a Unicode object on Python 2. Fixes issue reported in pull request #19.
- Issue #24: Added *DCCConnection.send_bytes* for transmitting binary data. *privmsg* remains to support transmitting text.

8.4

19 Jul 2013

- Code base now runs natively on Python 2 and Python 3, but requires six to be installed.
- Issue #25: Rate-limiting has been updated to be finer grained (preventing bursts exceeding the limit following idle periods).

8.3.2

17 Jul 2013

- Issue #22: Catch error in bot.py on NAMREPLY when nick is not in any visible channel.

8.3.1

04 Jul 2013

- Fixed encoding errors in server on Python 3.

8.3

22 Apr 2013

- Added a `set_keepalive` method to the ServerConnection. Sends a periodic PING message every indicated interval.

8.2

14 Apr 2013

- Added support for throttling send_raw messages via the ServerConnection object. For example, on any connection object:

  ```
  connection.set_rate_limit(30)
  ```

  That would set the rate limit to 30 Hz (30 per second). Thanks to Jason Kendall for the suggestion and bug fixes.

8.1.2

12 Apr 2013

- Fix typo in *client.NickMask*.

8.1.1

04 Apr 2013

- Fix typo in bot.py.

8.1

02 Apr 2013

- Issue #15: Added client support for ISUPPORT directives on server connections. Now, each ServerConnection has a *features* attribute which reflects the features supported by the server. See the docs for *irc.features* for details about the implementation.

8.0.1

11 Feb 2013

- Issue #14: Fix errors when handlers of the same priority are added under Python 3. This also fixes the unintended behavior of allowing handlers of the same priority to compare as unequal.

8.0

12 Jan 2013

This release brings several backward-incompatible changes to the scheduled commands.

- Refactored implementation of schedule classes. No longer do they override the datetime constructor, but now only provide suitable classmethods for construction in various forms.
- Removed backward-compatible references from irc.client.
- Remove 'arguments' parameter from scheduled commands.

Clients that reference the schedule classes from irc.client or that construct them from the basic constructor will need to update to use the new class methods:

```
- DelayedCommand -> DelayedCommand.after
- PeriodicCommand -> PeriodicCommand.after
```

Arguments may no longer be passed to the 'function' callback, but one is encouraged instead to use functools.partial to attach parameters to the callback. For example:

```
DelayedCommand.after(3, func, ('a', 10))
```

becomes:

```
func = functools.partial(func, 'a', 10)
DelayedCommand.after(3, func)
```

This mode puts less constraints on the both the handler and the caller. For example, a caller can now pass keyword arguments instead:

```
func = functools.partial(func, name='a', quantity=10)
DelayedCommand.after(3, func)
```

Readability, maintainability, and usability go up.

7.1.2

12 Jan 2013

- Issue #13: TypeError on Python 3 when constructing PeriodicCommand (and thus execute_every).

7.1.1

05 Jan 2013

- Fixed regression created in 7.0 where PeriodicCommandFixedDelay would only cause the first command to be scheduled, but not subsequent ones.

7.1

04 Jan 2013

- Moved scheduled command classes to irc.schedule module. Kept references for backwards-compatibility.

7.0

01 Jan 2013

- PeriodicCommand now raises a ValueError if it's created with a negative or zero delay (meaning all subsequent commands are immediately due). This fixes #12.
- Renamed the parameters to the IRC object. If you use a custom event loop and your code constructs the IRC object with keyword parameters, you will need to update your code to use the new names, so:

```
IRC(fn_to_add_socket=adder, fn_to_remove_socket=remover, fn_to_add_
↪timeout=timeout)
```

  becomes:

```
IRC(on_connect=adder, on_disconnect=remover, on_schedule=timeout)
```

  If you don't use a custom event loop or you pass the parameters positionally, no change is necessary.

6.0.1

29 Dec 2012

- Fixed some unhandled exceptions in server client connections when the client would disconnect in response to messages sent after select was called.

6.0

28 Dec 2012

- Moved *LineBuffer* and *DecodingLineBuffer* from client to buffer module. Backward-compatible references have been kept for now.
- Removed daemon mode and log-to-file options for server.
- Miscellaneous bugfixes in server.

5.1.1

27 Dec 2012

- Fix error in 2to3 conversion on irc/server.py (issue #11).

5.1

25 Dec 2012

The IRC library is now licensed under the MIT license.

- Added irc/server.py, based on hircd by Ferry Boender.
- Added support for CAP command (pull request #10), thanks to Danneh Oaks.

5.0

15 Nov 2012

Another backward-incompatible change. In irc 5.0, many of the unnecessary getter functions have been removed and replaced with simple attributes. This change addresses issue #2. In particular:

- Connection._get_socket() -> Connection.socket (including subclasses)
- Event.eventtype() -> Event.type
- Event.source() -> Event.source
- Event.target() -> Event.target
- Event.arguments() -> Event.arguments

The *nm_to_** functions were removed. Instead, use the NickMask class attributes.

These deprecated function aliases were removed from irc.client:

```
- parse_nick_modes -> modes.parse_nick_modes
- parse_channel_modes -> modes.parse_channel_modes
- generated_events -> events.generated
- protocol_events -> events.protocol
- numeric_events -> events.numeric
- all_events -> events.all
- irc_lower -> strings.lower
```

Also, the parameter name when constructing an event was renamed from *eventtype* to simply *type*.

4.0

15 Nov 2012

- Removed deprecated arguments to ServerConnection.connect. See notes on the 3.3 release on how to use the connect_factory parameter if your application requires ssl, ipv6, or other connection customization.

3.6.1

15 Nov 2012

- Filter out disconnected sockets when processing input.

3.6

08 Nov 2012

- Created two new exceptions in *irc.client*: *MessageTooLong* and *InvalidCharacters*.
- Use explicit exceptions instead of ValueError when sending data.

3.5

06 Nov 2012

- SingleServerIRCBot now accepts keyword arguments which are passed through to the *ServerConnection.connect* method. One can use this to use SSL for connections:

```
factory = irc.connection.Factory(wrapper=ssl.wrap_socket)
bot = irc.bot.SingleServerIRCBot(..., connect_factory = factory)
```

3.4.2

25 Oct 2012

- Issue #6: Fix AttributeError when legacy parameters are passed to *ServerConnection.connect*.
- Issue #7: Fix TypeError on *iter(LineBuffer)*.

3.4.1

22 Oct 2012

3.4 never worked - the decoding customization feature was improperly implemented and never tested.

- The ServerConnection now allows custom classes to be supplied to customize the decoding of incoming lines. For example, to disable the decoding of incoming lines, replace the *buffer_class* on the ServerConnection with a version that passes through the lines directly:

```
irc.client.ServerConnection.buffer_class = irc.client.LineBuffer
```

  This fixes #5.

3.4

18 Oct 2012

Broken Release

3.3

17 Oct 2012

- Added *connection* module with a Factory for creating socket connections.
- Added *connect_factory* parameter to the ServerConnection.

It's now possible to create connections with custom SSL parameters or other socket wrappers. For example, to create a connection with a custom SSL cert:

```
import ssl
import irc.client
import irc.connection
import functools

irc = irc.client.IRC()
server = irc.server()
wrapper = functools.partial(ssl.wrap_socket, ssl_cert=my_cert())
server.connect(connect_factory = irc.connection.Factory(wrapper=wrapper))
```

With this release, many of the parameters to *ServerConnection.connect* are now deprecated:

- localaddress
- localport
- ssl
- ipv6

Instead, one should pass the appropriate values to a *connection.Factory* instance and pass that factory to the .connect method. Backwards-compatibility will be maintained for these parameters until the release of irc 4.0.

3.2.3

14 Oct 2012

- Restore Python 2.6 compatibility.

3.2.2

11 Oct 2012

- Protect from UnicodeDecodeError when decoding data on the wire when data is not properly encoded in ASCII or UTF-8.

3.2.1

11 Oct 2012

- Additional branch protected by mutex.

3.2

09 Oct 2012

- Implemented thread safety via a reentrant lock guarding shared state in IRC objects.

3.1.1

07 Oct 2012

- Fix some issues with bytes/unicode on Python 3

3.1

07 Oct 2012

- Distribute using setuptools rather than paver.
- Minor tweaks for Python 3 support. Now installs on Python 3.

3.0.1

29 Sep 2012

- Added error checking when sending a message - for both message length and embedded carriage returns. Fixes #4.
- Updated README.

3.0

05 Sep 2012

- Improved Unicode support. Fixes failing tests and errors lowering Unicode channel names.
- Issue #3541414 - The ServerConnection and DCCConnection now encode any strings as UTF-8 before transmitting.
- Issue #3527371 - Updated strings.FoldedCase to support comparison against objects of other types.
- Shutdown the sockets before closing.

Applications that are currently encoding unicode as UTF-8 before passing the strings to *ServerConnection.send_raw* need to be updated to send Unicode or ASCII.

2.0.4

05 Sep 2012

This release officially deprecates 2.0.1-2.0.3 in favor of 3.0.

- Re-release of irc 2.0 (without the changes from 2.0.1-2.0.3) for correct compatibility indication.

2.0

22 May 2012

- DelayedCommands now use the local time for calculating 'at' and 'due' times. This will be more friendly for simple servers. Servers that expect UTC times should either run in UTC or override DelayedCommand.now to return an appropriate time object for 'now'. For example:

```
def startup_bot():
    irc.client.DelayedCommand.now = irc.client.DelayedCommand.utcnow
    ...
```

1.1

11 May 2012

- Added irc.client.PeriodicCommandFixedDelay. Schedule this command to have a function executed at a specific time and then at periodic intervals thereafter.

1.0

10 May 2012

- Removed *irclib* and *ircbot* legacy modules.

0.9

02 May 2012

- Fix file saving using dccreceive.py on Windows. Fixes #2863199.
- Created NickMask class from nm_to_* functions. Now if a source is a NickMask, one can access the .nick, .host, and .user attributes.
- Use correct attribute for saved connect args. Fixes #3523057.

0.8

24 Apr 2012

- Added ServerConnection.reconnect method. Fixes #3515580.

0.7.1

24 Apr 2012

- Added missing events. Fixes #3515578.

0.7

18 Apr 2012

- Moved functionality from irclib module to irc.client module.
- Moved functionality from ircbot module to irc.bot module.
- Retained irclib and ircbot modules for backward-compatibility. These will be removed in 1.0.
- Renamed project to simply 'irc'.

To support the new module structure, simply replace references to the irclib module with irc.client and ircbot module with irc.bot. This project will support that interface through all versions of irc 1.x, so if you've made these changes, you can safely depend on *irc >= 0.7, <2.0dev*.

0.6.3

16 Apr 2012

- Fixed failing test where DelayedCommands weren't being sorted properly. DelayedCommand a now subclass of the DateTime object, where the command's due time is the datetime. Fixed issue #3518508.

0.6.2

15 Apr 2012

- Fixed incorrect usage of Connection.execute_delayed (again).

0.6.0

- Minimum Python requirement is now Python 2.6. Python 2.3 and earlier should use 0.5.0 or earlier.
- Removed incorrect usage of Connection.execute_delayed. Added Connection.execute_every. Fixed issue 3516241.
- Use new-style classes.

CHAPTER 2

irc package

Subpackages

irc.tests package

Submodules

irc.tests.test_bot module

irc.tests.test_client module

irc.tests.test_schedule module

Module contents

Submodules

irc.bot module

Simple IRC bot library.

This module contains a single-server IRC bot class that can be used to write simpler bots.

class `irc.bot.`**`Channel`**
Bases: `object`

A class for keeping information about an IRC channel.

`add_user`(*nick*)

`admins`()
Returns an unsorted list of the channel's admins.

change_nick(*before*, *after*)

clear_mode(*mode*, *value=None*)
Clear mode on the channel.

Arguments:

mode – The mode (a single-character string).

value – Value

halfops()
Returns an unsorted list of the channel's half-operators.

has_allow_external_messages()

has_key()

has_limit()

has_mode(*mode*)

has_topic_lock()

has_user(*nick*)
Check whether the channel has a user.

is_admin(*nick*)
Check whether a user has admin status in the channel.

is_halfop(*nick*)
Check whether a user has half-operator status in the channel.

is_invite_only()

is_moderated()

is_oper(*nick*)
Check whether a user has operator status in the channel.

is_owner(*nick*)
Check whether a user has owner status in the channel.

is_protected()

is_secret()

is_voiced(*nick*)
Check whether a user has voice mode set in the channel.

limit()

opers()
Returns an unsorted list of the channel's operators.

owners()
Returns an unsorted list of the channel's owners.

remove_user(*nick*)

set_mode(*mode*, *value=None*)
Set mode on the channel.

Arguments:

mode – The mode (a single-character string).

value – Value

set_userdetails(*nick*, *details*)

user_dicts

user_modes = 'ovqha'
Modes which are applicable to individual users, and which should be tracked in the mode_users dictionary.

users()
Returns an unsorted list of the channel's users.

voiced()
Returns an unsorted list of the persons that have voice mode set in the channel.

class irc.bot.**ExponentialBackoff**(**attrs*)
Bases: *irc.bot.ReconnectStrategy*

A ReconnectStrategy implementing exponential backoff with jitter.

check()

max_interval = 300

min_interval = 60

run(*bot*)

class irc.bot.**ReconnectStrategy**
Bases: object

An abstract base class describing the interface used by SingleServerIRCBot for handling reconnect following disconnect events.

run(*bot*)
Invoked by the bot on disconnect. Here a strategy can determine how to react to a disconnect.

class irc.bot.**ServerSpec**(*host*, *port=6667*, *password=None*)
Bases: object

An IRC server specification.

```
>>> spec = ServerSpec('localhost')
>>> spec.host
'localhost'
>>> spec.port
6667
>>> spec.password
```

```
>>> spec = ServerSpec('127.0.0.1', 6697, 'fooP455')
>>> spec.password
'fooP455'
```

class irc.bot.**SingleServerIRCBot**(*server_list*, *nickname*, *realname*, *reconnection_interval=<object object>*, *recon=<irc.bot.ExponentialBackoff object>*, ***connect_params*)
Bases: *irc.client.SimpleIRCClient*

A single-server IRC bot class.

The bot tries to reconnect if it is disconnected.

The bot keeps track of the channels it has joined, the other clients that are present in the channels and which of those that have operator or voice modes. The "database" is kept in the self.channels attribute, which is an IRCDict of Channels.

Arguments:

server_list – A list of ServerSpec objects or tuples of parameters suitable for constructing ServerSpec objects. Defines the list of servers the bot will use (in order).

nickname – The bot's nickname.

realname – The bot's realname.

recon – A ReconnectStrategy for reconnecting on disconnect or failed connection.

dcc_connections – A list of initiated/accepted DCC connections.

****connect_params – parameters to pass through to the connect** method.

die(*msg='Bye, cruel world!'*)
Let the bot die.

Arguments:

msg – Quit message.

disconnect(*msg="I'll be back!"*)
Disconnect the bot.

The bot will try to reconnect after a while.

Arguments:

msg – Quit message.

get_version()
Returns the bot version.

Used when answering a CTCP VERSION request.

jump_server(*msg='Changing servers'*)
Connect to a new server, possibly disconnecting from the current.

The bot will skip to next server in the server_list each time jump_server is called.

on_ctcp(*c*, *e*)
Default handler for ctcp events.

Replies to VERSION and PING requests and relays DCC requests to the on_dccchat method.

on_dccchat(*c*, *e*)

start()
Start the bot.

irc.client module

Internet Relay Chat (IRC) protocol client library.

This library is intended to encapsulate the IRC protocol in Python. It provides an event-driven IRC client framework. It has a fairly thorough support for the basic IRC protocol, CTCP, and DCC chat.

To best understand how to make an IRC client, the reader more or less must understand the IRC specifications. They are available here: [IRC specifications].

The main features of the IRC client framework are:

- Abstraction of the IRC protocol.

- Handles multiple simultaneous IRC server connections.
- Handles server PONGing transparently.
- Messages to the IRC server are done by calling methods on an IRC connection object.
- Messages from an IRC server triggers events, which can be caught by event handlers.
- Reading from and writing to IRC server sockets are normally done by an internal select() loop, but the select()ing may be done by an external main loop.
- Functions can be registered to execute at specified times by the event-loop.
- Decodes CTCP tagging correctly (hopefully); I haven't seen any other IRC client implementation that handles the CTCP specification subtilties.
- A kind of simple, single-server, object-oriented IRC client class that dispatches events to instance methods is included.

Current limitations:

- Data is not written asynchronously to the server, i.e. the write() may block if the TCP buffers are stuffed.
- DCC file transfers are not supported.
- RFCs 2810, 2811, 2812, and 2813 have not been considered.

Notes:

- connection.quit() only sends QUIT to the server.
- ERROR from the server triggers the error event and the disconnect event.
- dropping of the connection triggers the disconnect event.

class irc.client.**Connection**(*reactor*)
Bases: object

Base class for IRC connections.

execute_at(*at*, *function*, *arguments=()*)

execute_delayed(*delay*, *function*, *arguments=()*)

execute_every(*period*, *function*, *arguments=()*)

socket
The socket for this connection

class irc.client.**DCCConnection**(*reactor*, *dcctype*)
Bases: *irc.client.Connection*

A DCC (Direct Client Connection).

DCCConnection objects are instantiated by calling the dcc method on a Reactor object.

connect(*address*, *port*)
Connect/reconnect to a DCC peer.

Arguments: address – Host/IP address of the peer.

port – The port number to connect to.

Returns the DCCConnection object.

disconnect(*message=''*)
Hang up the connection and close the object.

Arguments:

message – Quit message.

listen()
Wait for a connection/reconnection from a DCC peer.

Returns the DCCConnection object.

The local IP address and port are available as self.localaddress and self.localport. After connection from a peer, the peer address and port are available as self.peeraddress and self.peerport.

privmsg(*text*)
Send text to DCC peer.

The text will be padded with a newline if it's a DCC CHAT session.

process_data()
[Internal]

send_bytes(*bytes*)
Send data to DCC peer.

socket = None

exception irc.client.**DCCConnectionError**
Bases: *irc.client.IRCError*

class irc.client.**Event**(*type*, *source*, *target*, *arguments=None*, *tags=None*)
Bases: object

An IRC event.

```
>>> print(Event('privmsg', '@somebody', '#channel'))
type: privmsg, source: @somebody, target: #channel, arguments: [], tags: []
```

exception irc.client.**IRCError**
Bases: Exception

An IRC exception

exception irc.client.**InvalidCharacters**
Bases: ValueError

Invalid characters were encountered in the message

exception irc.client.**MessageTooLong**
Bases: ValueError

Message is too long

class irc.client.**NickMask**
Bases: str

A nickmask (the source of an Event)

```
>>> nm = NickMask('pinky!username@example.com')
>>> nm.nick
'pinky'
```

```
>>> nm.host
'example.com'
```

```
>>> nm.user
'username'
```

```
>>> isinstance(nm, six.text_type)
True
```

```
>>> nm = '!red@yahoo.ru'
>>> if not six.PY3: nm = nm.decode('utf-8')
>>> nm = NickMask(nm)
```

```
>>> isinstance(nm.nick, six.text_type)
True
```

Some messages omit the userhost. In that case, None is returned.

```
>>> nm = NickMask('irc.server.net')
>>> nm.nick
'irc.server.net'
>>> nm.userhost
>>> nm.host
>>> nm.user
```

classmethod from_group(*group*)

classmethod from_params(*nick*, *user*, *host*)

host

nick

user

userhost

class irc.client.**PrioritizedHandler**
Bases: irc.client.Base

class irc.client.**Reactor**(*on_connect=<function Reactor.__do_nothing>*, *on_disconnect=<function Reactor.__do_nothing>*)
Bases: object

Processes events from one or more IRC server connections.

This class implements a reactor in the style of the reactor pattern.

When a Reactor object has been instantiated, it can be used to create Connection objects that represent the IRC connections. The responsibility of the reactor object is to provide an event-driven framework for the connections and to keep the connections alive. It runs a select loop to poll each connection's TCP socket and hands over the sockets with incoming data for processing by the corresponding connection.

The methods of most interest for an IRC client writer are server, add_global_handler, remove_global_handler, process_once, and process_forever.

This is functionally an event-loop which can either use it's own internal polling loop, or tie into an external event-loop, by having the external event-system periodically call *process_once* on the instantiated reactor class. This will allow the reactor to process any queued data and/or events.

Calling *process_forever* will hand off execution to the reactor's internal event-loop, which will not return for the life of the reactor.

Here is an example:

client = irc.client.Reactor() server = client.server() server.connect("irc.some.where", 6667, "my_nickname") server.privmsg("a_nickname", "Hi there!") client.process_forever()

This will connect to the IRC server irc.some.where on port 6667 using the nickname my_nickname and send the message "Hi there!" to the nickname a_nickname.

The methods of this class are thread-safe; accesses to and modifications of its internal lists of connections, handlers, and delayed commands are guarded by a mutex.

add_global_handler(*event*, *handler*, *priority=0*)
Adds a global handler function for a specific event type.

Arguments:

event – Event type (a string). Check the values of numeric_events for possible event types.

handler – Callback function taking 'connection' and 'event' parameters.

priority – A number (the lower number, the higher priority).

The handler function is called whenever the specified event is triggered in any of the connections. See documentation for the Event class.

The handler functions are called in priority order (lowest number is highest priority). If a handler function returns "NO MORE", no more handlers will be called.

dcc(*dcctype='chat'*)
Creates and returns a DCCConnection object.

Arguments:

dcctype – "chat" for DCC CHAT connections or "raw" for DCC SEND (or other DCC types). If "chat", incoming data will be split in newline-separated chunks. If "raw", incoming data is not touched.

disconnect_all(*message=''*)
Disconnects all connections.

execute_at(*at*, *function*, *arguments=()*)
Execute a function at a specified time.

Arguments:

at – Execute at this time (a standard Unix timestamp). function – Function to call. arguments – Arguments to give the function.

execute_delayed(*delay*, *function*, *arguments=()*)
Execute a function after a specified time.

delay – How many seconds to wait. function – Function to call. arguments – Arguments to give the function.

execute_every(*period*, *function*, *arguments=()*)
Execute a function every 'period' seconds.

period – How often to run (always waits this long for first). function – Function to call. arguments – Arguments to give the function.

process_data(*sockets*)
Called when there is more data to read on connection sockets.

Arguments:

sockets – A list of socket objects.

See documentation for Reactor.__init__.

process_forever(*timeout=0.2*)
Run an infinite loop, processing data from connections.

This method repeatedly calls process_once.

Arguments:

timeout – Parameter to pass to process_once.

process_once(*timeout=0*)
Process data from connections once.

Arguments:

timeout – How long the select() call should wait if no data is available.

This method should be called periodically to check and process incoming data, if there are any. If that seems boring, look at the process_forever method.

process_timeout()
Called when a timeout notification is due.

See documentation for Reactor.__init__.

remove_global_handler(*event*, *handler*)
Removes a global handler function.

Arguments:

event – Event type (a string). handler – Callback function.

Returns 1 on success, otherwise 0.

scheduler_class
alias of `DefaultScheduler`

server()
Creates and returns a ServerConnection object.

sockets

class irc.client.**ServerConnection**(*reactor*)
Bases: *irc.client.Connection*

An IRC server connection.

ServerConnection objects are instantiated by calling the server method on a Reactor object.

action(*target*, *action*)
Send a CTCP ACTION command.

add_global_handler(**args*)
Add global handler.

See documentation for IRC.add_global_handler.

admin(*server=''*)
Send an ADMIN command.

as_nick(*name*)
Set the nick for the duration of the context.

buffer_class
alias of `DecodingLineBuffer`

cap(*subcommand*, **args*)
Send a CAP command according to the spec.

Arguments:

subcommand – LS, LIST, REQ, ACK, CLEAR, END args – capabilities, if required for given subcommand

Example:

.cap('LS') .cap('REQ', 'multi-prefix', 'sasl') .cap('END')

close()
Close the connection.

This method closes the connection permanently; after it has been called, the object is unusable.

connect(*server*, *port*, *nickname*, *password=None*, *username=None*, *ircname=None*, *connect_factory=<irc.connection.Factory object>*)
Connect/reconnect to a server.

Arguments:

- server - Server name
- port - Port number
- nickname - The nickname
- password - Password (if any)
- username - The username
- ircname - The IRC name ("realname")
- server_address - The remote host/port of the server
- connect_factory - A callable that takes the server address and returns a connection (with a socket interface)

This function can be called to reconnect a closed connection.

Returns the ServerConnection object.

ctcp(*ctcptype*, *target*, *parameter=''*)
Send a CTCP command.

ctcp_reply(*target*, *parameter*)
Send a CTCP REPLY command.

disconnect(*message=''*)
Hang up the connection.

Arguments:

message – Quit message.

get_nickname()
Get the (real) nick name.

This method returns the (real) nickname. The library keeps track of nick changes, so it might not be the nick name that was passed to the connect() method.

get_server_name()
Get the (real) server name.

This method returns the (real) server name, or, more specifically, what the server calls itself.

globops(*text*)
Send a GLOBOPS command.

info(*server=''*)
Send an INFO command.

invite(*nick*, *channel*)
Send an INVITE command.

is_connected()
Return connection status.

Returns true if connected, otherwise false.

ison(*nicks*)
Send an ISON command.

Arguments:

nicks – List of nicks.

join(*channel*, *key=''*)
Send a JOIN command.

kick(*channel*, *nick*, *comment=''*)
Send a KICK command.

links(*remote_server=''*, *server_mask=''*)
Send a LINKS command.

list(*channels=None*, *server=''*)
Send a LIST command.

lusers(*server=''*)
Send a LUSERS command.

mode(*target*, *command*)
Send a MODE command.

motd(*server=''*)
Send an MOTD command.

names(*channels=None*)
Send a NAMES command.

nick(*newnick*)
Send a NICK command.

notice(*target*, *text*)
Send a NOTICE command.

oper(*nick*, *password*)
Send an OPER command.

part(*channels*, *message=''*)
Send a PART command.

pass_(*password*)
Send a PASS command.

ping(*target*, *target2=''*)
Send a PING command.

pong(*target*, *target2=''*)
Send a PONG command.

privmsg(*target*, *text*)
Send a PRIVMSG command.

privmsg_many(*targets*, *text*)
Send a PRIVMSG command to multiple targets.

process_data()
read and process input from self.socket

quit(*message=''*)
Send a QUIT command.

reconnect()
Reconnect with the last arguments passed to self.connect()

remove_global_handler(**args*)
Remove global handler.

See documentation for IRC.remove_global_handler.

send_items(**items*)
Send all non-empty items, separated by spaces.

send_raw(*string*)
Send raw string to the server.

The string will be padded with appropriate CR LF.

set_keepalive(*interval*)
Set a keepalive to occur every `interval` on this connection.

set_rate_limit(*frequency*)
Set a *frequency* limit (messages per second) for this connection. Any attempts to send faster than this rate will block.

socket = None

squit(*server*, *comment=''*)
Send an SQUIT command.

stats(*statstype*, *server=''*)
Send a STATS command.

time(*server=''*)
Send a TIME command.

topic(*channel*, *new_topic=None*)
Send a TOPIC command.

trace(*target=''*)
Send a TRACE command.

user(*username*, *realname*)
Send a USER command.

userhost(*nicks*)
Send a USERHOST command.

users(*server=''*)
Send a USERS command.

version(*server=''*)
Send a VERSION command.

wallops(*text*)
Send a WALLOPS command.

who(*target=''*, *op=''*)
Send a WHO command.

whois(*targets*)
Send a WHOIS command.

whowas(*nick*, *max=''*, *server=''*)
Send a WHOWAS command.

exception irc.client.**ServerConnectionError**
Bases: *irc.client.IRCError*

exception irc.client.**ServerNotConnectedError**
Bases: *irc.client.ServerConnectionError*

class irc.client.**SimpleIRCClient**
Bases: object

A simple single-server IRC client class.

This is an example of an object-oriented wrapper of the IRC framework. A real IRC client can be made by subclassing this class and adding appropriate methods.

The method on_join will be called when a "join" event is created (which is done when the server sends a JOIN messsage/command), on_privmsg will be called for "privmsg" events, and so on. The handler methods get two arguments: the connection object (same as self.connection) and the event object.

Functionally, any of the event names in *events.py* may be subscribed to by prefixing them with *on_*, and creating a function of that name in the child-class of *SimpleIRCClient*. When the event of *event_name* is received, the appropriately named method will be called (if it exists) by runtime class introspection.

See *_dispatcher()*, which takes the event name, postpends it to *on_*, and then attemps to look up the class member function by name and call it.

Instance attributes that can be used by sub classes:

reactor – The Reactor instance.

connection – The ServerConnection instance.

dcc_connections – A list of DCCConnection instances.

connect(**args*, ***kwargs*)
Connect using the underlying connection

dcc_connect(*address*, *port*, *dcctype='chat'*)
Connect to a DCC peer.

Arguments:

address – IP address of the peer.

port – Port to connect to.

Returns a DCCConnection instance.

dcc_listen(*dcctype='chat'*)
Listen for connections from a DCC peer.

Returns a DCCConnection instance.

reactor_class
alias of *Reactor*

start()
Start the IRC client.

irc.client.**ip_numstr_to_quad**(*num*)
Convert an IP number as an integer given in ASCII representation to an IP address string.

```
>>> ip_numstr_to_quad('3232235521')
'192.168.0.1'
>>> ip_numstr_to_quad(3232235521)
'192.168.0.1'
```

irc.client.**ip_quad_to_numstr**(*quad*)
Convert an IP address string (e.g. '192.168.0.1') to an IP number as a base-10 integer given in ASCII representation.

```
>>> ip_quad_to_numstr('192.168.0.1')
'3232235521'
```

irc.client.**is_channel**(*string*)
Check if a string is a channel name.

Returns true if the argument is a channel name, otherwise false.

irc.connection module

class irc.connection.**Factory**(*bind_address=None*, *wrapper=<function <lambda>>*, *ipv6=False*)
Bases: object

A class for creating custom socket connections.

To create a simple connection:

server_address = ('localhost', 80) Factory()(server_address)

To create an SSL connection:

Factory(wrapper=ssl.wrap_socket)(server_address)

To create an SSL connection with parameters to wrap_socket:

wrapper = functools.partial(ssl.wrap_socket, ssl_cert=get_cert()) Factory(wrapper=wrapper)(server_address)

To create an IPv6 connection:

Factory(ipv6=True)(server_address)

Note that Factory doesn't save the state of the socket itself. The caller must do that, as necessary. As a result, the Factory may be re-used to create new connections with the same settings.

connect(*server_address*)

family = 2

irc.connection.**identity**(*x*)

irc.ctcp module

Handle Client-to-Client protocol per the best available spec.

irc.ctcp.**dequote**(*message*)
Dequote a message according to CTCP specifications.

The function returns a list where each element can be either a string (normal message) or a tuple of one or two strings (tagged messages). If a tuple has only one element (ie is a singleton), that element is the tag; otherwise the tuple has two elements: the tag and the data.

Arguments:

message – The message to be decoded.

irc.dict module

class irc.dict.**IRCDict**(**args*, ***kargs*)
Bases: jaraco.collections.KeyTransformingDict

A dictionary of names whose keys are case-insensitive according to the IRC RFC rules.

```
>>> d = IRCDict({'[This]': 'that'}, A='foo')
```

The dict maintains the original case:

```
>>> '[This]' in ''.join(d.keys())
True
```

But the keys can be referenced with a different case

```
>>> d['a'] == 'foo'
True
```

```
>>> d['{this}'] == 'that'
True
```

```
>>> d['{THIS}'] == 'that'
True
```

```
>>> '{thiS]' in d
True
```

This should work for operations like delete and pop as well.

```
>>> d.pop('A') == 'foo'
True
>>> del d['{This}']
>>> len(d)
0
```

static transform_key(*key*)

irc.events module

irc.features module

class irc.features.**FeatureSet**

Bases: object

An implementation of features as loaded from an ISUPPORT server directive.

Each feature is loaded into an attribute of the same name (but lowercased to match Python sensibilities).

```
>>> f = FeatureSet()
>>> f.load(['target', 'PREFIX=(abc)+-/', 'your message sir'])
>>> f.prefix == {'+': 'a', '-': 'b', '/': 'c'}
True
```

Order of prefix is relevant, so it is retained.

```
>>> tuple(f.prefix)
('+', '-', '/')
```

```
>>> f.load_feature('CHANMODES=foo,bar,baz')
>>> f.chanmodes
['foo', 'bar', 'baz']
```

load(*arguments*)

Load the values from the a ServerConnection arguments

load_feature(*feature*)

remove(*feature_name*)

set(*name*, *value=True*)

set a feature value

irc.features.**string_int_pair**(*target*, *sep=':'*)

irc.functools module

irc.functools.**save_method_args**(*method*)

Wrap a method such that when it is called, the args and kwargs are saved on the method.

```
>>> class MyClass(object):
...     @save_method_args
...     def method(self, a, b):
...         print(a, b)
>>> my_ob = MyClass()
>>> my_ob.method(1, 2)
1 2
>>> my_ob._saved_method.args
(1, 2)
>>> my_ob._saved_method.kwargs
{}
>>> my_ob.method(a=3, b='foo')
3 foo
>>> my_ob._saved_method.args
```

```
()
>>> my_ob._saved_method.kwargs == dict(a=3, b='foo')
True
```

The arguments are stored on the instance, allowing for different instance to save different args.

```
>>> your_ob = MyClass()
>>> your_ob.method({str('x'): 3}, b=[4])
{'x': 3} [4]
>>> your_ob._saved_method.args
({'x': 3},)
>>> my_ob._saved_method.args
()
```

irc.modes module

irc.modes.**parse_channel_modes**(*mode_string*)

Parse a channel mode string.

The function returns a list of lists with three members: sign, mode and argument. The sign is "+" or "-". The argument is None if mode isn't one of "b", "k", "l", "v", "o", "h", or "q".

Example:

```
>>> parse_channel_modes("+ab-c foo")
[['+', 'a', None], ['+', 'b', 'foo'], ['-', 'c', None]]
```

irc.modes.**parse_nick_modes**(*mode_string*)

Parse a nick mode string.

The function returns a list of lists with three members: sign, mode and argument. The sign is "+" or "-". The argument is always None.

Example:

```
>>> parse_nick_modes("+ab-c")
[['+', 'a', None], ['+', 'b', None], ['-', 'c', None]]
```

irc.rfc module

irc.rfc.**clean_pages**()

irc.rfc.**get_pages**(*filename*)

irc.rfc.**remove_footer**(*page*)

irc.rfc.**remove_header**(*page*)

irc.rfc.**save_clean**()

irc.schedule module

class irc.schedule.**DefaultScheduler**

Bases: tempora.schedule.InvokeScheduler, *irc.schedule.IScheduler*

execute_after(*delay*, *func*)

execute_at(*when*, *func*)

execute_every(*period*, *func*)

class irc.schedule.**IScheduler**
Bases: object

execute_after(*delay*, *func*)
execute func after delay

execute_at(*when*, *func*)
execute func at when

execute_every(*period*, *func*)
execute func every period

run_pending()
invoke the functions that are due

irc.server module

irc/server.py

This server has basic support for:

- Connecting
- Channels
- Nicknames
- Public/private messages

It is MISSING support for notably:

- Server linking
- Modes (user and channel)
- Proper error reporting
- Basically everything else

It is mostly useful as a testing tool or perhaps for building something like a private proxy on. Do NOT use it in any kind of production code or anything that will ever be connected to by the public.

class irc.server.**IRCChannel**(*name*, *topic='No topic'*)
Bases: object

An IRC channel.

class irc.server.**IRCClient**(*request*, *client_address*, *server*)
Bases: socketserver.BaseRequestHandler

IRC client connect and command handling. Client connection is handled by the handle method which sets up a two-way communication with the client. It then handles commands sent by the client by dispatching them to the handle_ methods.

exception Disconnect
Bases: BaseException

IRCClient.**client_ident**()
Return the client identifier as included in many command replies.

IRCClient.**finish**()
The client conection is finished. Do some cleanup to ensure that the client doesn't linger around in any channel or the client list, in case the client didn't properly close the connection with PART and QUIT.

IRCClient.**handle**()

IRCClient.**handle_dump**(*params*)
Dump internal server information for debugging purposes.

IRCClient.**handle_join**(*params*)
Handle the JOINing of a user to a channel. Valid channel names start with a # and consist of a-z, A-Z, 0-9 and/or '_'.

IRCClient.**handle_nick**(*params*)
Handle the initial setting of the user's nickname and nick changes.

IRCClient.**handle_part**(*params*)
Handle a client parting from channel(s).

IRCClient.**handle_ping**(*params*)
Handle client PING requests to keep the connection alive.

IRCClient.**handle_privmsg**(*params*)
Handle sending a private message to a user or channel.

IRCClient.**handle_quit**(*params*)
Handle the client breaking off the connection with a QUIT command.

IRCClient.**handle_topic**(*params*)
Handle a topic command.

IRCClient.**handle_user**(*params*)
Handle the USER command which identifies the user to the server.

exception irc.server.**IRCError**(*code*, *value*)
Bases: Exception

Exception thrown by IRC command handlers to notify client of a server/client error.

classmethod **from_name**(*name*, *value*)

class irc.server.**IRCServer**(**args*, ***kwargs*)
Bases: socketserver.ThreadingMixIn, socketserver.TCPServer

allow_reuse_address = True

channels = {}
Existing channels (IRCChannel instances) by channel name

clients = {}
Connected clients (IRCClient instances) by nick name

daemon_threads = True

irc.server.**get_args**()

irc.server.**main**()

irc.strings module

class irc.strings.**IRCFoldedCase**

Bases: jaraco.text.FoldedCase

A version of FoldedCase that honors the IRC specification for lowercased strings (RFC 1459).

```
>>> IRCFoldedCase('Foo^').lower()
'foo~'
```

```
>>> IRCFoldedCase('[this]') == IRCFoldedCase('{THIS}')
True
```

```
>>> IRCFoldedCase().lower()
''
```

lower()

translation = {65: 97, 66: 98, 67: 99, 68: 100, 69: 101, 70: 102, 71: 103, 72: 104, 73: 105, 74: 106, 75: 107, 76: 108, 77:

irc.strings.**lower**(*str*)

Module contents

Full-featured Python IRC library for Python.

- Project home
- Docs
- History

CHAPTER 3

License

License is indicated in the project metadata (typically one or more of the Trove classifiers). For more details, see this explanation.

CHAPTER 4

Overview

This library provides a low-level implementation of the IRC protocol for Python. It provides an event-driven IRC client framework. It has a fairly thorough support for the basic IRC protocol, CTCP, and DCC connections.

In order to understand how to make an IRC client, it's best to read up first on the IRC specifications.

CHAPTER 5

Installation

IRC requires Python versions specified in the download pages and definitely supports Python 3.

You have several options to install the IRC project.

- Use `easy_install irc` or `pip install irc` to grab the latest version from the cheeseshop (recommended).
- Run `python setup.py install` (from the source distribution).

CHAPTER 6

Client Features

The main features of the IRC client framework are:

- Abstraction of the IRC protocol.
- Handles multiple simultaneous IRC server connections.
- Handles server PONGing transparently.
- Messages to the IRC server are done by calling methods on an IRC connection object.
- Messages from an IRC server triggers events, which can be caught by event handlers.
- Reading from and writing to IRC server sockets is normally done by an internal `select()` loop, but the `select()` may be done by an external main loop.
- Functions can be registered to execute at specified times by the event-loop.
- Decodes CTCP tagging correctly (hopefully); I haven't seen any other IRC client implementation that handles the CTCP specification subtilties.
- A kind of simple, single-server, object-oriented IRC client class that dispatches events to instance methods is included.
- DCC connection support.

Current limitations:

- The IRC protocol shines through the abstraction a bit too much.
- Data is not written asynchronously to the server (and DCC peers), i.e. the `write()` may block if the TCP buffers are stuffed.
- Like most projects, documentation is lacking ...

Unfortunately, this library isn't as well-documented as I would like it to be. I think the best way to get started is to read and understand the example program `irccat`, which is included in the distribution.

The following files might be of interest:

- `irc/client.py`

  The library itself. Read the code along with comments and docstrings to get a grip of what it does. Use it at your own risk and read the source, Luke!

- `irc/bot.py`

  An IRC bot implementation.

- `irc/server.py`

  A basic IRC server implementation. Suitable for testing, but not production quality.

CHAPTER 7

Examples

Example scripts in the scripts directory:

- `irccat`

  A simple example of how to use the IRC client. `irccat` reads text from stdin and writes it to a specified user or channel on an IRC server.

- `irccat2`

  The same as above, but using the `SimpleIRCClient` class.

- `servermap`

  Another simple example. `servermap` connects to an IRC server, finds out what other IRC servers there are in the net and prints a tree-like map of their interconnections.

- `testbot`

  An example bot that uses the `SingleServerIRCBot` class from `irc.bot`. The bot enters a channel and listens for commands in private messages or channel traffic. It also accepts DCC invitations and echos back sent DCC chat messages.

- `dccreceive`

  Receives a file over DCC.

- `dccsend`

  Sends a file over DCC.

NOTE: If you're running one of the examples on a unix command line, you need to escape the `#` symbol in the channel. For example, use `\\#test` or `"#test"` instead of `#test`.

CHAPTER 8

Scheduling Events

The library includes a default event Scheduler as `irc.schedule.DefaultScheduler`, but this scheduler can be replaced with any other scheduler. For example, to use the schedule package, include it in your dependencies and install it into the IRC library as so:

```
class ScheduleScheduler(irc.schedule.IScheduler):
    def execute_every(self, period, func): schedule.every(period).do(func)
    def execute_at(self, when, func): schedule.at(when).do(func)
    def execute_after(self, delay, func): raise NotImplementedError("Not supported")
    def run_pending(self): schedule.run_pending()
irc.client.Reactor.scheduler_class = ScheduleScheduler
```

CHAPTER 9

Decoding Input

By default, the IRC library attempts to decode all incoming streams as UTF-8, even though the IRC spec stipulates that no specific encoding can be expected. Since assuming UTF-8 is not reasonable in the general case, the IRC library provides options to customize decoding of input by customizing the `ServerConnection` class. The `buffer_class` attribute on the `ServerConnection` determines which class is used for buffering lines from the input stream, using the `buffer` module in jaraco.stream. By default it is `buffer.DecodingLineBuffer`, but may be re-assigned with another class, following the interface of `buffer.LineBuffer`. The `buffer_class` attribute may be assigned for all instances of `ServerConnection` by overriding the class attribute.

For example:

```
from jaraco.stream import buffer
irc.client.ServerConnection.buffer_class = buffer.LenientDecodingLineBuffer
```

The `LenientDecodingLineBuffer` attempts UTF-8 but falls back to latin-1, which will avoid `UnicodeDecodeError` in all cases (but may produce unexpected behavior if an IRC user is using another encoding).

The buffer may be overridden on a per-instance basis (as long as it's overridden before the connection is established):

```
server = irc.client.IRC().server()
server.buffer_class = buffer.LenientDecodingLineBuffer
server.connect()
```

Alternatively, some clients may still want to decode the input using a different encoding. To decode all input as latin-1 (which decodes any input), use the following:

```
irc.client.ServerConnection.buffer_class.encoding = 'latin-1'
```

Or decode to UTF-8, but use a replacement character for unrecognized byte sequences:

```
irc.client.ServerConnection.buffer_class.errors = 'replace'
```

Or, to simply ignore all input that cannot be decoded:

```
class IgnoreErrorsBuffer(buffer.DecodingLineBuffer):
    def handle_exception(self):
        pass
irc.client.ServerConnection.buffer_class = IgnoreErrorsBuffer
```

On Python 2, it was possible to use the `buffer.LineBuffer` itself, which will pass the raw bytes. On Python 3, the library requires text for message processing, so a decoding buffer must be used. Therefore, use of the `LineBuffer` is considered deprecated and not supported on Python 3. Clients should use one of the above techniques for decoding input to text.

CHAPTER 10

Notes and Contact Info

Enjoy.

Maintainer: Jason R. Coombs <jaraco@jaraco.com>

Original Author: Joel Rosdahl <joel@rosdahl.net>

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index