Hanyuu-sama Documentation

Release 1.3

R/a/dio

March 04, 2013

CONTENTS

Contents:

CHAPTER

ONE

HANYUU

1.1 hanyuu Package

1.1.1 `hanyuu` Package

1.1.2 `config` Module

hanyuu.config.**get**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

hanyuu.config.**getfloat**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

hanyuu.config.**getint**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

hanyuu.config.**has_option**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

hanyuu.config.**has_section**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

hanyuu.config.**items**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

hanyuu.config.**load_configuration**(*filepaths*)
Creates a new `ConfigParser.SafeConfigParser` and tries parsing all `filepaths` given.

`filepaths` should be a list of filenames.

Returns nothing, instead assigns itself to the global variable *configuration* and abstracts itself by calling `create_abstractions()`

hanyuu.config.**options**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

hanyuu.config.**reload_configuration**()
Creates a new `ConfigParser.SafeConfigParser` and passes it the same filenames as given in the last call to `load_configuration()`.

This effectively 'reloads' the configuration files.

hanyuu.config.**sections**(**args*, ***kwargs*)
See `ConfigParser.RawConfigParser`

1.1.3 `utils` Module

class `hanyuu.utils.`**`Switch`**(*initial*, *timeout=15*)
Bases: `object`

A timed switch. Evaluates truthy if the time has expired, else falsy.

`reset`(*timeout=15*)

1.1.4 Subpackages

abstractions Package

`abstractions` Package

package to abstract the database from the rest of the code.

For users: submodules are grouped by their overarcing thema such as DJ profiles and users in the same submodule, track information in the same submodule, AFK streamer information in the same submodule, etc.

For developers: submodules should be of the grouping type where closely related data structures are placed together in a module.

`tracks` Module

class `hanyuu.abstractions.tracks.`**`Length`**
Bases: `int`

A simple subclass of `int` to support formatting on it without having to know the exact format or value in the rest of the code.

`format`()

Returns unicode A formatted [hh:]mm:nn string of the integer.

exception `hanyuu.abstractions.tracks.`**`NoTrackEntry`**
Bases: `exceptions.Exception`

Raised when a `Song` instance accesses Track only attributes without having an audio file attached to it.

class `hanyuu.abstractions.tracks.`**`Plays`**(*song*, *sequence*)
Bases: `list`

A simple subclass of `list` to support some extra attributes.

This class is returned when you access `Track.plays` and is a collection of play times of the `Track` in question.

The collection contains `datetime.datetime` objects or objects that act the same as such with extra methods (for future additions).

`add`(*time*, *dj=None*)
Adds a played entry to the `Track` object.

The exact time it was played at. :params time: A `datetime.datetime` instance.

The DJ that played this track at the time. :params dj: A `hanyuu.abstractions.users.DJ` instance.

Returns None

Note: It's good practice to add the DJ argument to all the code already.

The current database however ignores this argument.

---

last

Returns The time that last occured.

Return type `datetime.datetime` object.

remove(*time*, *dj=None*)
Removes a played entry from the `Track` object.

The time this was played at. :params time: A `datetime.datetime` instance.

The DJ that played this track at the time. If this is `None` it will be ignored otherwise it will be used for exact matching. :params dj: A `hanyuu.abstractions.users.DJ` instance.

Returns None

Note: Currently the *dj* argument is completely ignored.

---

save()
Saves changes to the database.

class `hanyuu.abstractions.tracks.`**Requests**
Bases: `list`

A simple subclass of `list` to support some extra attributes.

last

Returns The time that last occured.

Return type `datetime.datetime` object.

class `hanyuu.abstractions.tracks.`**Track**(*meta*, ***kwargs*)
Bases: `object`

An instance of a known track in our database. This can also be used for adding new tracks.

A 'known' track is one we have seen before. This means there is no difference between tracks we have an audio file of and ones we only know metadata of. The object easily allows you to check if it has a corresponding audio file available or not.

artist

Returns The artist of this song.

Return type unicode

filename

Returns unicode The filename of the audio file.

Raises `NoTrackEntry` if the song has no audio file.

Note: This is relative to the configured *media.directory* configuration.

---

filepath

Returns unicode The full path to the audio file.

Raises NoTrackEntry if the song has no audio file.

classmethod from_esong_id(*id*)
Returns an instance based on the *esong* table ID.

> **Warning:** Don't use this method in production code.

classmethod from_track_id(*id*)
Returns an instance based on the *tracks* table ID.

> **Warning:** Don't use this method in production code.

length

Returns Length of the song or 0 if none available.

Return type Length

Note: The song length is only 100% accurate if the song has an audio file available. Otherwise it's an approximation from when it was last played.

---

metadata

Returns A metadata string of '[artist -] title' where artist is optional

Return type unicode

Note: This uses the *tracks* table if available before trying the other table.

---

open(**args*, ***kwargs*)
Opens the associated file and returns a file object.

This handles the path finding for you.

Params unicode mode The mode to be passed to the open() call.

Returns An open file object.

Raises NoTrackEntry if the song has no audio file.

plays

Returns A mutable object with all the playing data in it.

Return type Plays

requests

Returns A mutable objects with all request data in it.

Return type Requests

Raises NoTrackEntry if the song has no audio file.

save()
Saves all the changes done so far on this object into the database.

Note: This method can do multiple queries to the database depending on the changes done on the object.

---

title

Returns The title of this song.

Return type unicode

hanyuu.abstractions.tracks.**create_metadata_string**(*track*)
Creates a '[artist -] title' string of the hanyuu.db.models.Track instance.

hanyuu.abstractions.tracks.**requires_track**(*func*)
Decorator that raises NoTrackEntry if the song instance has no associated audio file in the database.

Currently this only checks if *self._track* is falsy.

users Module

A module used for the abstractions of the users part of the database.

class hanyuu.abstractions.users.**DJ**(*id=None*, *name=None*)
Bases: object

Encapsulates the concept of a DJ in our system.

This abstracts the database from the rest of the code. But does return a database related object since it's a simple one.

classmethod resolve_id(*id*)
Resolves a DJ identifier to a DJ username.

Returns a class instance or raises DoesNotExist

classmethod resolve_name(*name*)
Resolves a DJ username to a DJ identifier.

Returns an integer that is the DJ identifier or 0 if the DJ username does not exist.

db Package

db Package

common Module

legacy Module

models Module

class hanyuu.db.models.**Base**(**args*, ***kwargs*)
Bases: peewee.Model

Simple base class to inherit from so all the other models inherit the database connection used.

DoesNotExist
alias of BaseDoesNotExist

id = <peewee.PrimaryKeyField object at 0x3204590>

class hanyuu.db.models.**DJ**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *djs* table.

DoesNotExist
alias of DJDoesNotExist

css = <peewee.CharField object at 0x3204850>

description = <peewee.TextField object at 0x3204750>

id = <peewee.PrimaryKeyField object at 0x3204650>

image = <peewee.TextField object at 0x3204790>

name = <peewee.CharField object at 0x3204710>

priority = <peewee.IntegerField object at 0x3204810>

queue

user

visible = <peewee.IntegerField object at 0x32047d0>

class hanyuu.db.models.**Fave**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *efave* table.

DoesNotExist
alias of FaveDoesNotExist

id = <peewee.PrimaryKeyField object at 0x3252b50>

nickname = <peewee.ForeignKeyField object at 0x3252d10>

song = <peewee.ForeignKeyField object at 0x3252d50>

class hanyuu.db.models.**LastFm**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *lastfm* table.

DoesNotExist
alias of LastFmDoesNotExist

id = <peewee.PrimaryKeyField object at 0x32520d0>

nick = <peewee.CharField object at 0x3252190>

username = <peewee.CharField object at 0x32521d0>

class hanyuu.db.models.**NickRequest**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *nickrequesttime* table.

DoesNotExist
alias of NickRequestDoesNotExist

host = <peewee.TextField object at 0x3204ed0>

id = <peewee.PrimaryKeyField object at 0x3204c90>

time = <peewee.DateTimeField object at 0x3204f10>

class hanyuu.db.models.**Nickname**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *enick* table.

DoesNotExist
alias of NicknameDoesNotExist

authcode = <peewee.CharField object at 0x3252490>

dtb = <peewee.DateTimeField object at 0x3252450>

faves

first_seen = <peewee.DateTimeField object at 0x3252410>

id = <peewee.PrimaryKeyField object at 0x3252250>

nickname = <peewee.CharField object at 0x32523d0>

class hanyuu.db.models.**Play**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *eplay* table.

DoesNotExist
alias of PlayDoesNotExist

id = <peewee.PrimaryKeyField object at 0x3252890>

song = <peewee.ForeignKeyField object at 0x3252a90>

time = <peewee.DateTimeField object at 0x3252ad0>

class hanyuu.db.models.**Queue**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the new design *queue* table.

DoesNotExist
alias of QueueDoesNotExist

dj = <peewee.ForeignKeyField object at 0x3255990>

id = <peewee.PrimaryKeyField object at 0x32553d0>

ip = <peewee.TextField object at 0x3255950>

song = <peewee.ForeignKeyField object at 0x32558d0>

time = <peewee.DateTimeField object at 0x3255890>

track = <peewee.ForeignKeyField object at 0x3255910>

type = <peewee.IntegerField object at 0x3255850>

class hanyuu.db.models.**Relay**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *relays* table.

DoesNotExist
alias of RelayDoesNotExist

active = <peewee.IntegerField object at 0x3255fd0>

base_name = <peewee.CharField object at 0x3255dd0>

bitrate = <peewee.IntegerField object at 0x3255e90>

country = <peewee.CharField object at 0x3256090>

disabled = <peewee.IntegerField object at 0x32560d0>

format = <peewee.CharField object at 0x3255ed0>

id = <peewee.PrimaryKeyField object at 0x3255a10>

listener_limit = <peewee.IntegerField object at 0x3255f90>

listeners = <peewee.IntegerField object at 0x3255f50>

mountpoint = <peewee.CharField object at 0x3255e50>

owner = <peewee.CharField object at 0x3255d90>

passcode = <peewee.CharField object at 0x3256050>

port = <peewee.IntegerField object at 0x3255e10>

priority = <peewee.IntegerField object at 0x3255f10>

subdomain = <peewee.CharField object at 0x3255d50>

class hanyuu.db.models.**Song**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *esong* table.

DoesNotExist
alias of SongDoesNotExist

faves

classmethod from_meta(*metadata*)
Returns the first match found of metadata

Params unicode metadata A string of metadata.

Returns Song instance.

Raises Song.DoesNotExist if no result was found.

Note: This currently does no pre-fetching of the faves and plays

---

hash = <peewee.CharField object at 0x3252750>

hash_link = <peewee.CharField object at 0x3252810>

id = <peewee.PrimaryKeyField object at 0x3252690>

length = <peewee.IntegerField object at 0x3252790>

meta = <peewee.TextField object at 0x32527d0>

plays

classmethod query_from_meta(*metadata*)
Returns the first match found of metadata

Params unicode metadata A string of metadata.

Returns peewee.SelectQuery instance.

Note: This currently does no pre-fetching of the faves and plays

---

queued

class hanyuu.db.models.**Track**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *tracks* table.

DoesNotExist
alias of TrackDoesNotExist

acceptor = <peewee.CharField object at 0x3255210>

album = <peewee.CharField object at 0x3255090>

artist = <peewee.CharField object at 0x3252fd0>

filename = <peewee.TextField object at 0x32550d0>

classmethod from_meta(*metadata*)
Returns the first match found of metadata

Params unicode metadata A string of metadata.

Returns Track instance.

hash = <peewee.CharField object at 0x3255290>

id = <peewee.PrimaryKeyField object at 0x3252dd0>

last_editor = <peewee.CharField object at 0x3255250>

last_played = <peewee.DateTimeField object at 0x3255150>

last_requested = <peewee.DateTimeField object at 0x3255190>

needs_reupload = <peewee.IntegerField object at 0x3255350>

priority = <peewee.IntegerField object at 0x32552d0>

queued

request_count = <peewee.IntegerField object at 0x3255310>

search_tags = <peewee.TextField object at 0x3255110>

title = <peewee.CharField object at 0x3255050>

usable = <peewee.IntegerField object at 0x32551d0>

class hanyuu.db.models.**User**(**args*, ***kwargs*)
Bases: hanyuu.db.models.Base

Models the legacy *users* table.

DoesNotExist
alias of UserDoesNotExist

dj = <peewee.ForeignKeyField object at 0x3204bd0>

id = <peewee.PrimaryKeyField object at 0x32048d0>

name = <peewee.CharField object at 0x3204b50>

password = <peewee.CharField object at 0x3204b90>

privileges = <peewee.IntegerField object at 0x3204c10>

ircbot Package

ircbot Package

commands Module

Subpackages

irclib Package

irclib Package

connection Module

class hanyuu.ircbot.irclib.connection.**Connection**(*irclibobj*)

Bases: object

Base class for IRC connections.

Must be overridden.

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

Executes a function at a specified time.

See Also:

session.Session.execute_at()

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

Executes a function after a specified number of seconds.

See Also:

session.Session.execute_delayed()

class hanyuu.ircbot.irclib.connection.**Event**(*eventtype*, *source*, *target*, *arguments=None*)

Bases: hanyuu.ircbot.irclib.connection.Event

Class representing an IRC event.

exception hanyuu.ircbot.irclib.connection.**IRCError**

Bases: exceptions.Exception

Represents an IRC exception.

class hanyuu.ircbot.irclib.connection.**ServerConnection**(*irclibobj*)

Bases: hanyuu.ircbot.irclib.connection.Connection

This class represents an IRC server connection.

action(*target*, *action*)

Send a CTCP ACTION command.

admin(*server=u''*)

Send an ADMIN command.

close()

Close the connection.

> **Warning:** 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*, *localaddress=u''*, *localport=0*, *ssl=False*, *ipv6=False*, *encoding=u'utf-8'*)
Connect/reconnect to a server.

Parameters

- **server** – Server name.
- **port** – Port number.
- **nickname** – The nickname.
- **password** – IRC Password (if any).

Note: This is NOT the NickServ password; you have to send that manually.

- **username** – The IRC username.
- **ircname** – The IRC name ('realname').
- **localaddress** – Bind the connection to a specific local IP address.
- **localport** – Bind the connection to a specific local port.
- **ssl** – Enable support for ssl.
- **ipv6** – Enable support for ipv6.

This function can be called to reconnect a closed connection.

Returns the ServerConnection object.

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

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

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

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.

get_topic(*channel*)
Return the topic of a channel.

Note: You must be joined to the channel in order to get the topic.

---

globops(*text*)
Send a GLOBOPS command.

hasaccess(*channel*, *nick*)
Check if nick is halfop or higher

hasanymodes(*channel*, *nick*, *modes*)
Check if nick has any of the specified modes on a channel.

inchannel(*channel*, *nick*)
Check if nick is in channel

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

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

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

Returns True if the argument is a channel name, otherwise False.

is_connected()
Return connection status.

Returns True if connected, otherwise False.

is_identified(*nick*)
Checks if a user has identified for their Nickname via NickServ returns boolean

ishop(*channel*, *nick*)
Check if nick is half operator on a channel.

isnormal(*channel*, *nick*)
Check if nick is a normal on a channel.

ison(*nicks*)
Send an ISON command.

isop(*channel*, *nick*)
Check if nick is operator or higher on a channel.

isvoice(*channel*, *nick*)
Check if nick has voice on a channel.

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

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

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

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

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

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

motd(*server=u''*)
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=u''*)
Send a PART command.

pass_(*password*)
Send a PASS command.

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

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

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

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

process_data()
Processes incoming data and dispatches handlers.

Only for internal use.

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

Note: This is not the same as `disconnect()`.

Note: Many IRC servers don't use your quit message unless you've been connected for at least 5 minutes.

---

reconnect(*message=u''*)
Disconnect and connect with same parameters

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

The string will be padded with appropriate CR LF.

send_raw_instant(*string*)
Send raw string to the server, bypassing the flood protection.

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

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

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

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

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

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

userhost (*nicks*)
Send a USERHOST command.

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

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

wallops (*text*)
Send a WALLOPS command.

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

whois (*targets*)
Send a WHOIS command.

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

exception hanyuu.ircbot.irclib.connection.**ServerConnectionError**
Bases: hanyuu.ircbot.irclib.connection.IRCError

exception hanyuu.ircbot.irclib.connection.**ServerNotConnectedError**
Bases: hanyuu.ircbot.irclib.connection.ServerConnectionError

dcc Module

class hanyuu.ircbot.irclib.dcc.**DCCConnection** (*irclibobj*, *dcctype*, *dccinfo=(None*, *0)*)
Bases: hanyuu.ircbot.irclib.connection.Connection

This class represents a DCC connection.

DCCConnection objects are instantiated by calling session.Session.dcc().

For usage, see connect() and listen().

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

Parameters

- **address** – Host/IP address of the peer.
- **port** – The port number to connect to.

Returns the DCCConnection object.

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

Parameters **message** – Quit message.

Note: After calling this method, the object becomes unusable.

---

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

Returns the DCCConnection object.

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

privmsg(*string*)
Send data to DCC peer.

The string will be padded with appropriate LF if it's a DCC CHAT session.

process_data()
[Internal]

exception hanyuu.ircbot.irclib.dcc.**DCCConnectionError**
Bases: hanyuu.ircbot.irclib.connection.IRCError

session Module A high level session object to the lower level irclib.connection module.

class hanyuu.ircbot.irclib.session.**HighEvent**(*server*, *command*, *nickname*, *channel*, *message*)
Bases: object

A abstracted event of the IRC library.

classmethod from_low_event(*server*, *low_event*)
Generates a high level event from a low level one.

class hanyuu.ircbot.irclib.session.**Nickname**(*host*, *nickname_only=False*)
Bases: object

A simple class that represents a nickname on IRC.

Contains information such as actual nickname, hostmask and more.

class hanyuu.ircbot.irclib.session.**Session**(*encoding=u'utf-8'*, *handle_ctcp=True*)
Class that handles one or several IRC server connections.

When a Session object has been instantiated, it can be used to create Connection objects that represent the IRC connections. The responsibility of the Session object is to provide a high-level 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. It then encapsulates the low level IRC events generated by the Connection objects into higher level versions.

ctcp_source = None
Used to respond to CTCP SOURCE messages.

ctcp_version = None
Used to respond to CTCP VERSION messages.

dcc(*dcctype=u'chat'*, *dccinfo=(None, 0)*)
Creates and returns a `dcc.DCCConnection` object.

Parameters **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=u''*)
Disconnects all connections.

Parameters **message** – The quit message to send to servers.

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

Parameters

- **at** – Time to execute at (standard "time_t" time).
- **function** – The function to call.
- **arguments** – Arguments to give the function.

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

Parameters

- **delay** – How many seconds to wait.
- **function** – The function to call.
- **arguments** – Arguments to give the function.

handlers = {}

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

Parameters **sockets** – A list of socket objects to be processed.

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

This method repeatedly calls `process_once()`.

Parameters **timeout** – Parameter to pass to process_once.

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

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

This method should be called periodically to check and process incoming and outgoing data, if there is any.

It calls `process_data()`, `_send_once()` and `process_timeout()`.

It will also examine when we last received data from the server; if it exceeds a specified time limit, the Session assumes that we have lost connection to the server and will attempt to reconnect us.

If calling it manually seems boring, look at the `process_forever()` method.

process_timeout()
This is called to process any delayed commands that are registered to the Session object.

See Also:

`process_once()`

register_socket(*socket*, *conn*)
Internal method used to map the sockets on `connection.Connection` to the connections themselves.

server()
Creates and returns a `connection.ServerConnection` object.

hanyuu.ircbot.irclib.session.**event_handler**(*events*, *channels=[]*, *nicks=[]*, *modes=u''*, *regex=u''*)
The decorator for high level event handlers. By decorating a function with this, the function is registered in the global `Session` event handler list, `Session.handlers`.

Parameters

- **events** – The events that the handler should subscribe to. This can be both a string and a list; if a string is provided, it will be added as a single element in a list of events. This rule applies to *channels* and *nicks* as well.
- **channels** – The channels that the events should trigger on. Given an empty list, all channels will trigger the event.
- **nicks** – The nicknames that this handler should trigger for. Given an empty list, all nicknames will trigger the event.
- **modes** – The required channel modes that are needed to trigger this event. If an empty mode string is specified, no modes are needed to trigger the event.
- **regex** – The event will only be triggered if the `HighEvent.message` matches the specified regex. If no regex is specified, any `HighEvent.message` will do.

`hanyuu.ircbot.irclib.session.`**high_level_events = [u'connect', u'text', u'join', u'part', u'kick', u'quit', u'mode**

All the high level events that we can register to. Low level events that aren't on this list can be registered to as well, but they will not be parsed.

`tracker` Module Module that contains all the classes required to track channels, nicknames modes and other related stuff.

class `hanyuu.ircbot.irclib.tracker.`**IRCTracker**

This class is used to track nicknames, channels, and the modes that are associated to nicknames on channels. It also tracks channel topics.

This tracker uses an internal Sqlite database to store its information.

add_mode(*chan*, *nick*, *mode*)

Sets 'mode' on 'nick' in the channel 'chan'.

close()

Closes the Sqlite connection.

execute(*query*)

Executes a Sqlite query and returns the results.

has_chan(*chan*)

Returns True if the tracker is familiar with the channel 'chan'.

has_modes(*chan*, *nick*, *modes*, *operator=u'and'*)

Returns true if the nickname 'nick' has the modes 'modes' in the channel 'chan'.

Based on the value of 'operator', the return value is different; if the operator is 'and', the nickname must have ALL of the specified modes. If the operator is 'or', the nickname must have ANY of the specified modes.

has_nick(*nick*)

Returns True if the tracker is familiar with the nickname 'nick'.

in_chan(*chan*, *nick*)

Returns true if the nickname 'nick' is in the channel 'chan'.

join(*chan*, *nick*)

Tells the tracker that the nickname 'nick' joined 'chan'.

nick(*nick*, *newnick*)

Tells the tracker that the nickname 'nick' has changed to 'newnick'.

part(*chan*, *nick*)
Tells the tracker that the nickname 'nick' left 'chan'.

quit(*nick*)
Tells the tracker that the nickname 'nick' has left the server.

rem_mode(*chan*, *nick*, *mode*)
Unsets 'mode' on 'nick' in the channel 'chan'

topic(*chan*, *topic=None*)
If 'topic' is None, this gets the topic in the channel 'chan'.

Otherwise, the topic will be set to 'topic'.

class hanyuu.ircbot.irclib.tracker.**SqliteCursor**(*conn*)
A simple Sqlite cursor.

utils Module

hanyuu.ircbot.irclib.utils.**ip_numstr_to_quad**(*num*)
Convert an IP number as an integer given in ASCII representation (e.g. '3232235521') to an IP address string (e.g. '192.168.0.1').

hanyuu.ircbot.irclib.utils.**ip_quad_to_numstr**(*quad*)
Convert an IP address string (e.g. '192.168.0.1') to an IP number as an integer given in ASCII representation (e.g. '3232235521').

hanyuu.ircbot.irclib.utils.**irc_lower**(*s*)
Returns a lowercased string.

The definition of lowercased comes from the IRC specification (RFC 1459).

hanyuu.ircbot.irclib.utils.**mask_matches**(*nick*, *mask*)
Check if a nick matches a mask.

Returns True if the nick matches, otherwise False.

hanyuu.ircbot.irclib.utils.**nick_characters = u'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU**
The characters that are permitted in IRC nicknames

hanyuu.ircbot.irclib.utils.**nm_to_h**(*s*)
Get the host part of a nickmask.

(The source of an connection.Event is a nickmask.)

hanyuu.ircbot.irclib.utils.**nm_to_n**(*s*)
Get the nick part of a nickmask.

(The source of an connection.Event is a nickmask.)

hanyuu.ircbot.irclib.utils.**nm_to_u**(*s*)
Get the user part of a nickmask.

(The source of an connection.Event is a nickmask.)

hanyuu.ircbot.irclib.utils.**nm_to_uh**(*s*)
Get the userhost part of a nickmask.

(The source of an connection.Event is a nickmask.)

listener Package

listener Package

requests Package

requests Package

hanyuu.requests.**songdelay**(*val*)
Gives the time delay in seconds for a specific song request count.

Subpackages

servers Package

servers Package

fastcgi Module

status Package

status Package

class hanyuu.status.**Base**
Bases: object

Simple base class that sets the attribute :attr:cache to a :class:memcache.Client ready to be used.

cache

class hanyuu.status.**Site**
Bases: hanyuu.status.Base

Object that encapsulates state of the website.

dj
Returns the current DJ that is live.

Returns a abstractions.users.DJ object.

thread
Returns the current thread URL.

Returns a unicode string or None

class hanyuu.status.**Stream**
Bases: hanyuu.status.Base

Wrapping class around the memcache server and variables relevant to the status of the streaming server.

current
Gets the current song metadata playing on the master server.

Returns a unicode object.

listeners
Returns the total amount of listeners as an integer.

This is the listeners combined from all relay servers.

online
Returns if the master server is online or not.

Returns a boolean type.

peak_listeners

class hanyuu.status.**Streamer**
Bases: hanyuu.status.Base

Object that encapsulates state of the AFK streamer.

requests_enabled
Returns a bool indicating if the AFK streamer accepts requests.

This is False if either Requests got disabled explicitely or the AFK streamer is not streaming at the moment.

hanyuu.status.**memcache_client**()
Returns a pylibmc.Client object.

streamstatus Module

streamer Package

streamer Package

afkstreamer Module

Subpackages

audio Package

audio Package

encoder Module

files Module

icecast Module

Subpackages

garbage Package

garbage Package

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

h