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.

Example usage:

```
>>> import time
>>> a = Switch(5)
>>> bool(a)
False
>>> time.sleep(5)
>>> bool(a)
True
```

reset(*timeout=15*)

hanyuu.utils.**fix_encoding**(*metadata*)
This function tries to guess the correct encoding of metadata

Currently this checks for the following encodings in order: UTF-8, strict mode SJIS, replace mode UTF-8, replace mode

---

Note: This function also calls an unconditional *metadata.strip()* before returning the new string.

Parameters **metadata** (*bytes*) – A sequence in an unknown encoding.

Returns unicode An unicode string.

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

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

Models the legacy *radvars* table.

DoesNotExist
alias of RadvarDoesNotExist

id = <peewee.PrimaryKeyField object at 0x245bcd0>

name = <peewee.CharField object at 0x245b790>

value = <peewee.TextField object at 0x245b6d0>

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

Models the legacy *streamstatus* table.

DoesNotExist
alias of StatusDoesNotExist

bitrate = <peewee.IntegerField object at 0x245bd90>

dj = <peewee.ForeignKeyField object at 0x245bbd0>

end_time = <peewee.IntegerField object at 0x245bed0>

id = <peewee.PrimaryKeyField object at 0x245bb90>

is_afk_stream = <peewee.IntegerField object at 0x245bdd0>

is_streamdesk = <peewee.IntegerField object at 0x245bd10>

last_set = <peewee.DateTimeField object at 0x245be50>

listeners = <peewee.IntegerField object at 0x245bd50>

now_playing = <peewee.CharField object at 0x245bb10>

start_time = <peewee.IntegerField object at 0x245be10>

track = <peewee.ForeignKeyField object at 0x245be90>

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 0x2178150>

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 0x21783d0>

description = <peewee.TextField object at 0x2178310>

id = <peewee.PrimaryKeyField object at 0x2178210>

image = <peewee.TextField object at 0x1fc5610>

name = <peewee.CharField object at 0x21782d0>

priority = <peewee.IntegerField object at 0x2178390>

queue

status

user

visible = <peewee.IntegerField object at 0x2178350>

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 0x217a6d0>

nickname = <peewee.ForeignKeyField object at 0x217a890>

song = <peewee.ForeignKeyField object at 0x217a8d0>

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 0x2178c10>

nick = <peewee.CharField object at 0x2178cd0>

username = <peewee.CharField object at 0x2178d10>

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 0x2178a50>

id = <peewee.PrimaryKeyField object at 0x2178810>

time = <peewee.DateTimeField object at 0x2178a90>

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 0x2178fd0>

dtb = <peewee.DateTimeField object at 0x2178f90>

faves

first_seen = <peewee.DateTimeField object at 0x2178f50>

id = <peewee.PrimaryKeyField object at 0x2178d90>

nickname = <peewee.CharField object at 0x2178f10>

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 0x217a410>

song = <peewee.ForeignKeyField object at 0x217a610>

time = <peewee.DateTimeField object at 0x217a650>

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 0x217c510>

id = <peewee.PrimaryKeyField object at 0x217af10>

ip = <peewee.TextField object at 0x217c4d0>

song = <peewee.ForeignKeyField object at 0x217c450>

time = <peewee.DateTimeField object at 0x217c410>

track = <peewee.ForeignKeyField object at 0x217c490>

type = <peewee.IntegerField object at 0x217c3d0>

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 0x217cb50>

base_name = <peewee.CharField object at 0x217c950>

bitrate = <peewee.IntegerField object at 0x217ca10>

country = <peewee.CharField object at 0x217cbd0>

disabled = <peewee.IntegerField object at 0x217cc10>

format = <peewee.CharField object at 0x217ca50>

id = <peewee.PrimaryKeyField object at 0x217c590>

listener_limit = <peewee.IntegerField object at 0x217cb10>

listeners = <peewee.IntegerField object at 0x217cad0>

mountpoint = <peewee.CharField object at 0x217c9d0>

owner = <peewee.CharField object at 0x217c910>

passcode = <peewee.CharField object at 0x217cb90>

port = <peewee.IntegerField object at 0x217c990>

priority = <peewee.IntegerField object at 0x217ca90>

subdomain = <peewee.CharField object at 0x217c8d0>

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 0x217a2d0>

hash_link = <peewee.CharField object at 0x217a390>

id = <peewee.PrimaryKeyField object at 0x217a210>

length = <peewee.IntegerField object at 0x217a310>

meta = <peewee.TextField object at 0x217a350>

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 0x217ad50>

album = <peewee.CharField object at 0x217abd0>

artist = <peewee.CharField object at 0x217ab50>

filename = <peewee.TextField object at 0x217ac10>

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 0x217add0>

id = <peewee.PrimaryKeyField object at 0x217a950>

last_editor = <peewee.CharField object at 0x217ad90>

last_played = <peewee.DateTimeField object at 0x217ac90>

last_requested = <peewee.DateTimeField object at 0x217acd0>

needs_reupload = <peewee.IntegerField object at 0x217ae90>

priority = <peewee.IntegerField object at 0x217ae10>

queued

request_count = <peewee.IntegerField object at 0x217ae50>

search_tags = <peewee.TextField object at 0x217ac50>

status

title = <peewee.CharField object at 0x217ab90>

usable = <peewee.IntegerField object at 0x217ad10>

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 0x2178750>

id = <peewee.PrimaryKeyField object at 0x2178450>

name = <peewee.CharField object at 0x21786d0>

password = <peewee.CharField object at 0x2178710>

privileges = <peewee.IntegerField object at 0x2178790>

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.

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

class hanyuu.listener.**Listener**
Bases: asynchat.async_chat

READING_DATA = 0

READING_HEADERS = 3

READING_META = 1

READING_METASIZE = 2

collect_incoming_data(*data*)

found_terminator()

handle_close()

parse_headers(*headers*)

shutdown()

hanyuu.listener.**shutdown**()

hanyuu.listener.**start**()

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

class `hanyuu.streamer.afkstreamer.`**Streamer**(*attributes*)
Bases: `object`

Top wrapper of the AFK Streamer. This gives out filenames and metadata to the underlying `audio` module.

close(*force=False*)
Stop the audio pipeline and disconnects from icecast.

connect(**args*, ***kwargs*)
Deprecated since version 1.2: use `start()`: instead.

connected
Returns True if the audio modules `audio.icecast` is currently connected. Else returns False.

shutdown (**args*, ***kwargs*)
Deprecated since version 1.2: use `close()`: instead.

start ()
Starts the audio pipeline and connects to icecast.

supply_song ()
Returns a tuple of (filename, metadata) to be played next.

Subpackages

audio Package

audio Package

class hanyuu.streamer.audio.**FileInformation** (*filename*, *metadata=None*)
Bases: object

A class that should be returned from the function passed to `Manager` for file discovery.

This is to make switching functions easier since the `Manager` doesn't need to know what format the function returns but only know that it returns a `FileInformation` instance instead.

class hanyuu.streamer.audio.**Handlers**
Bases: dict

class hanyuu.streamer.audio.**Manager** (*source*, *processors=None*, *options=None*, *handlers=None*)
Bases: object

A class that manages the audio pipeline. Each component gets a reference to the processor before it.

Note: This is a very cruel pipeline and has specifics to our needs and is in no way a generic implementation.

Nor does it have proper definitions of what should go out or into a processor.

The `Manager` expects that all registered processors have at least the following characteristics:

start(): Called when Manager.start() is called. This should initialize required components. The `Manager` expects that a call to *close* and *start* is close to equal of recreating the whole instance.

close(): Called when Manager.close() is called. This should close down the processor cleanly and if potential long running cleanups are to be done should use the `garbage` sub package shipped with the **audio** package.

__init__(): Called when the Manager instance is created. This should not start any state dependant parts, these should be done in the *start* method instead.

Gets passed one positional argument that is the previous processor in the chain. Or if the first processor read below.

Gets passed extra keyword arguments if specified in the class attribute *options*. Read more about this attribute below.

The current version expects the first specified processor to take a function as *source* argument. That can be called for the filepath of an audiofile. This first processor is responsible for opening it.

Note: This means the processor doesn't actually need to decode the file but that it is just expected to accept the function as *source*. What it does with the function is not important to the `Manager`.

The current version expects the last specified processor to have several methods available to be used by the `Manager`. These are:

`status()`: A method that is called when `status()` is called. This should return something of significants to the user.

`metadata()`: A method that accepts a single *unicode* argument. This is called whenever new metadata is found at the start of the processor chain.

`close`()
Calls the *close* method on all registered processor instances.

> **Warning:** Exceptions are propagated.

`get_source`()

Returns unicode A full file path to an audio file.

The value returned from `Manager.source()` is expected to be an `FileInformation` object. But there is one exception to this rule.

When `Manager.source()` returns a different type it will be used as the positional arguments to the `FileInformation` constructor by using the *FileInformation(*returntype)* syntax.

`processors` = [<class 'hanyuu.streamer.audio.files.FileSource'>, <class 'hanyuu.streamer.audio.encoder.Encoder'>, <cl
A list of processors that are instanced in order and are passed their previous friend as first argument.

`start`()
Calls the *start* method on all registered processor instances.

This method does nothing if a previous call to `start()` was successful but `close()` was not called in between the two calls.

> **Warning:** Exceptions are propagated.

`status`()
Calls the *status* method on the last processor in the chain.

If no method was found returns `False` instead.

`hanyuu.streamer.audio.`**`test_config`**(*password=None*)

`hanyuu.streamer.audio.`**`test_dir`**(*directory=u'/media/F/Music'*, *files=None*)

`encoder` Module A very simple encoder module.

The encoders currently supported are listed below:

- LAME MP3 encoder

Other formats would require a rework of this module to be more generic.

class `hanyuu.streamer.audio.encoder.`**`Encoder`**(*source*, *options*, *handlers*)
Bases: `object`

An Encoder class that handles the encoder subprocess underneath.

Right now this class only supports encoding by using a LAME mp3 encoder binary. Other formats are not supported.

`close`()
This calls the `EncoderInstance.close()` method on the `EncoderInstance`.

options = [(u'lame_settings', [u'–cbr', u'-b', u'192', u'–resample', u'44.1'])]

report_close()
This method is called by the EncoderInstance class when it gets closed or an error occurs in the instance. This should handle the case gracefully and even restart the instance if the close was unintentional by the user.

The method registers the EncoderInstance instance for garbage collection by the garbage module.

restart()
This method rather then restart, destroys and then recreates the underlying EncoderInstance instance.

settings = None
The settings for encoding to pass to lame as a list.

start()
This clears our *alive* flag and starts a new EncoderInstance instance by calling start_instance().

start_instance()
This method is responsible for creating and starting the EncoderInstance class instances.

This creates a new EncoderInstance instance and calls the EncoderInstance.start() method on it.

After the call to 'start' returns the new instance is assigned to instance.

class hanyuu.streamer.audio.encoder.**EncoderInstance**(*encoder_manager*)
Bases: object

Class that represents a subprocessed encoder.

This is a non-generic class as of now and could be made generic and supporting different kind of format encoders. As of yet this only supports the LAME binary encoder.

Note: This class is used internally and should never be instantiated directly by the user.

---

close()

read(*size=4096*, *timeout=10.0*)

run()

start()

switch_source(*new_source*)

write(*data*)

class hanyuu.streamer.audio.encoder.**GarbageInstance**(*item=None*)
Bases: hanyuu.streamer.audio.garbage.Garbage

A garbage object to be registered for EncoderInstance class instances.

collect()

hanyuu.streamer.audio.encoder.**LAME_BIN = u'lame'**
The path to the LAME binary. This can be just 'lame' on bash environments.

files Module Module that handles file access and decoding to PCM.

It uses python-audiotools for the majority of the work done.

exception hanyuu.streamer.audio.files.**AudioError**
Bases: exceptions.Exception

Exception raised when an error occurs in this module.

class hanyuu.streamer.audio.files.**AudioFile**(*filename*)
Bases: object

A Simple wrapper around the audiotools library.

This opens the filename given wraps the file in a PCMConverter that turns it into PCM of format 44.1kHz, Stereo, 24-bit depth.

close()
Registers self for garbage collection. This method does not close anything and only registers itself for colleciton.

progress(*current*, *total*)
Dummy progress function

read(*size=4096*, *timeout=0.0*)
Returns at most a string of size *size*.

The *timeout* argument is unused. But kept in for compatibility with other read methods in the *audio* module.

class hanyuu.streamer.audio.files.**FileSource**(*source*, *options*, *handlers*)
Bases: object

The FileSource class expects a function as source.

This function should return an absolute path to a supported audio file as an unicode or bytes object.

No options are supported for this class.

No handlers are supported for this class.

change_source()
Calls the source function and returns the result if not None.

close()

initialize()
Sets the initial source from the source function.

read(*size=4096*, *timeout=10.0*)

skip()

start()
Starts the source

class hanyuu.streamer.audio.files.**GarbageAudioFile**(*item=None*)
Bases: hanyuu.streamer.audio.garbage.Garbage

Garbage class of the AudioFile class

collect()
Tries to close the AudioFile resources when called.

icecast Module A module that adds some extra useful wrapping around the **libshout** library.

class hanyuu.streamer.audio.icecast.**Icecast**(*source*, *options*, *handlers*)
Bases: object

The `Icecast` class expects a source that returns encoded MP3 audio data. Use of other formats as of now is not supported.

The source requires the following attributes:

`read()`: A method that returns encoded MP3 audio data.

param size An `int` signifying the amount of bytes to read.

returns `bytes` containing the requested MP3 audio data.

We accept a single option that is a full fletched configuration for the underlying **libshout** library.

- **icecast_config:** A `dict` containing the configuration for **libshout**. For the exact contents of the dictionary see `IcecastConfig`.

The following handler hooks are supported by `Icecast`.

- **icecast_start(icecast):** Called when the `Icecast.start()` is called.

  param icecast `Icecast` instance.

- **icecast_close(icecast):** Called when the `Icecast.close()` is called.

  param icecast `Icecast` instance.

- **icecast_connect(icecast, options):** Called when `Icecast.connect()` is called.

  param icecast `Icecast` instance.

  param options `IcecastConfig` instance used.

- **icecast_metadata(icecast, metadata):** Called when metadata is send to the server.

  param icecast `Icecast` instance.

  param metadata `unicode` instance containing the metadata send.

`close`()
Closes the libshout object and tries to join the thread if we are not calling this from our own thread.

`connect`()
Connect the libshout object to the configured server.

`connected`()
Returns True if the libshout object is currently connected to an icecast server.

`connecting_timeout = 5.0`
The time to wait when we lose connection by cause of external behaviour.

`metadata`(*metadata*)

`options = [(u'icecast_config', {})]`

`read`(*size*, *timeout=None*)

`reboot_libshout`()
Internal method

Tries to recreate the libshout object.

`run`()

`setup_libshout`()
Internal method

Creates a libshout object and puts the configuration to use.

start()
Starts the thread that reads from source and feeds it to icecast.

switch_source(*new_source*)
Tries to change the source without disconnect from icecast.

class hanyuu.streamer.audio.icecast.**IcecastConfig**(*attributes=None*)
Bases: dict

Simple dict subclass that knows how to apply the keys to a libshout object.

The following dictionary items are supported:

- **host:** The hostname of the icecast server to connect to. (defaults to localhost)
- **port:** The port the icecast server is running on. (defaults to 8000)
- **user:** The icecast user.
- **password:** The password for the icecast server.
- **mount:** The mountpoint to connect to.
- **format:** The format we are going to stream in.

  Can be one of the following:

  - 0 = OGG encoded data.
  - 1 = MP3 encoded data.
- **protocol:** The protocol to use to connect to the icecast server.

  Can be one of the following:

  - 0 = HTTP protocol
  - 1 = XAUDIOCAST protocol
  - 2 = ICY protocol

  If you are unsure of which one to use, you are most likely after the HTTP protocol.
- **name:** An optional name to show on the icecast page.
- **url:** An optional URL to be placed on the icecast page.
- **genre:** An optional genre to be shown on the icecast page.
- **agent:** The useragent to connect with. (defaults to libshout/version)
- **description:** An optional small description to show on the icecast page.
- **charset:** An optional charset to use for sending metadata. (defaults to UTF-8)
- **public:** An optional boolean specifying if this stream is to be marked as public in icecast. (This affects indexing on external sites.)
- **dumpfile:** A file to dump the streamed data to.
- **audio_info:** Can be ignored, see libshout docs for info.
- **metadata:** Can be ignored, see libshout docs for info.

setup(*shout*)
Setup 'shout' configuration by setting attributes on the object.

'shout' is a pylibshout.Shout object.

exception hanyuu.streamer.audio.icecast.**IcecastError**
Bases: exceptions.Exception

Subpackages

garbage Package

garbage Package

class hanyuu.streamer.audio.garbage.**Collector**
Bases: object

add(*garbage*)

classmethod add_hook(*hook*)

info()
Returns a list of GarbageInfo objects containing information about current pending garbage.

instance = <hanyuu.streamer.audio.garbage.Collector object at 0x2778050>

run()

class hanyuu.streamer.audio.garbage.**Garbage**(*item=None*)
Bases: object

collect()
Gets called on each collection cycle.

Should return True if the garbage got cleaned up properly, False if it requires another collect in the next cycle.

collector = <hanyuu.streamer.audio.garbage.Collector object at 0x2778050>

class hanyuu.streamer.audio.garbage.**Singleton**(*mcs*, *name*, *bases*, *dict*)
Bases: type

CHAPTER

TWO

INDICES AND TABLES

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

PYTHON MODULE INDEX

h