csbot Documentation

Release alpha

#cs-york

Jun 13, 2017

Contents

Contents:

CHAPTER 1

How to write plugins

Anatomy of a plugin

Plugins are automatically discovered if they match the right pattern. They must

- subclass *csbot.plugin.Plugin*, and
- live under the package specified by `csbot.core.Bot.PLUGIN_PACKAGE` (`csbot.plugins` by default).

For example, a minimal plugin that does nothing might live in `csbot/plugins/nothing.py` and look like:

```
from csbot.plugin import Plugin

class Nothing(Plugin):
    pass
```

A plugin's name is its class name in lowercase[1] and must be unique, so plugin classes should be named meaningfully. Changing a plugin name will cause it to lose access to its associated configuration and database, so try not to do that unless you're prepared to migrate these things.

The vast majority of interaction with the outside world is through subscribing to events and registering commands.

Events

Root events are generated when the bot receives data from the IRC server, and further events may be generated while handling an event.

All events are represented by the *Event* class, which is a dictionary of event-related information with some additional helpful attributes. See *Events* for further information on the *Event* class and available events.

Events are hooked with the *Plugin.hook()* decorator. The decorated method will be called for every event that matches the specified *event_type*, with the event object as the only argument. For example, a basic logging plugin that prints sent and received data:

[1] This can be changed by overriding the `plugin_name()` class method if absolutely necessary.

```
class Logger(Plugin):
    @Plugin.hook('core.raw.sent')
    def sent(self, e):
        print('<-- ' + e['message'])

    @Plugin.hook('core.raw.received')
    def received(self, e):
        print('--> ' + e['message'])
```

A single handler can hook more than one event:

```
class MessagePrinter(Plugin):
    @Plugin.hook('core.message.privmsg')
    @Plugin.hook('core.message.notice')
    def got_message(self, e):
        """Print out all messages, ignoring if they were PRIVMSG or NOTICE."""
        print(e['message'])
```

Commands

Registering commands provides a more structured way for users to interact with a plugin. A command can be any unique, non-empty sequence of non-whitespace characters, and are invoked when prefixed with the bot's configured command prefix. Command events use the *CommandEvent* class, extending a `core.message.privmsg` *Event* and adding the *arguments()* method and the `command` and `data` items.

```
class CommandTest(Plugin):
    @Plugin.command('test')
    def hello(self, e):
        print(e['command'] + ' invoked with arguments ' + repr(e.arguments()))
```

A single handler can be registered for more than one command, e.g. to give aliases, and commands and hooks can be freely mixed.

```
class Friendly(Plugin):
    @Plugin.hook('core.channel.joined')
    @Plugin.command('hello')
    @Plugin.command('hi')
    def hello(self, e):
        e.protocol.msg(e['channel'], 'Hello, ' + nick(e['user']))
```

Responding: the `BotProtocol` object

In the above example the `Event.protocol` attribute was used to respond back to the IRC server. This attribute is an instance of `BotProtocol`, which subclasses twisted.words.protocols.irc.IRCClient for IRC protocol support. The documentation for IRCClient is the best place to find out what methods are supported when responding to an event or command.

Configuration

Basic string key/value configuration can be stored in an INI-style file. A plugin's *config* attribute is a shortcut to a configuration section with the same name as the plugin. The Python 3 `configparser` is used instead of the Python 2 `ConfigParser` because it supports the mapping access protocol, i.e. it acts like a dictionary in addition to supporting its own API.

An example of using plugin configuration:

```
class Say(Plugin):
    @Plugin.command('say')
    def say(self, e):
        if self.config.getboolean('shout', False):
            e.protocol.msg(e['reply_to'], e['data'].upper() + '!')
        else:
            e.protocol.msg(e['reply_to'], e['data'])
```

For even more convenience, automatic fallback values are supported through the *CONFIG_DEFAULTS* attribute when using the *config_get()* or *config_getboolean()* methods instead of the corresponding methods on *config*. This is encouraged, since it makes it clear what configuration the plugin supports and what the default values are by looking at just one part of the plugin source code. The above example would look like this:

```
class Say(Plugin):
    CONFIG_DEFAULTS = {
        'shout': False,
    }

    @Plugin.command('say')
    def say(self, e):
        if self.config_getboolean('shout'):
            e.protocol.msg(e['reply_to'], e['data'].upper() + '!')
        else:
            e.protocol.msg(e['reply_to'], e['data'])
```

Configuration can be changed at runtime, but won't be saved. This allows for temporary state changes, whilst ensuring the startup state of the bot reflects the configuration file. For example, the above plugin could be modified with a toggle for the "shout" mode:

```
class Say(Plugin):
    # ...
    @Plugin.command('toggle')
    def toggle(self, e):
        self.config['shout'] = not self.config_get('shout')
```

Database

The bot supports easy access to MongoDB through PyMongo. Plugins have a `db` attribute which is a `pymongo.database.Database`, unique to the plugin and created as needed. Refer to the PyMongo documentation for further guidance on using the API.

CHAPTER 2

Events

All events are represented by *Event* instances. Every event has the following attributes:

`Event`**.bot = None**
The *Bot* which triggered the event.

`Event`**.event_type = None**
The name of the event.

`Event`**.datetime = None**
The value of `datetime.datetime.now()` when the event was triggered.

Event instances are also dictionaries, and the keys present depend on the particular event type. The following sections describe each event, specified as `event_type(keys)`.

Raw events

These events are very low-level and most plugins shouldn't need them.

core.raw.connected
Client established connection.

core.raw.disconnected
Client lost connection.

core.raw.sent(message)
Client sent *message* to the server.

core.raw.received(message)
Client received *message* from the server.

Bot events

These events represent changes in the bot's state.

core.self.connected
IRC connection successfully established.

core.self.joined(channel)
Client joined *channel*.

core.self.left(channel)
Client left *channel*.

Message events

These events occur when messages are received by the bot.

core.message.privmsg(channel, user, message, is_private, reply_to)
Received *message* from *user* which was sent to *channel*. If the message was sent directly to the client, i.e. *channel* is the client's nick and not a channel name, then *is_private* will be True and any response should be to *user*, not *channel*. *reply_to* is the channel/user any response should be sent to.

core.message.notice(channel, user, message, is_private, reply_to)
As `core.message.privmsg`, but representing a NOTICE rather than a PRIVMSG. Bear in mind that according to RFC 1459 "automatic replies must never be sent in response to a NOTICE message" - this definitely applies to bot functionality!

core.message.action(channel, user, message, is_private, reply_to)
Received a `CTCP ACTION` of *message* from *user* sent to *channel*. Other arguments are as for `core.message.privmsg`.

Channel events

These events occur when something about the channel changes, e.g. people joining or leaving, the topic changing, etc.

core.channel.joined(channel, user)
user joined *channel*.

core.channel.left(channel, user)
user left *channel*.

core.channel.names(channel, names, raw_names)
Received the list of users currently in the channel, in response to a `NAMES` command.

cores.channel.topic(channel, author, topic)
Fired whenever the channel topic is changed, and also immediately after joining a channel. The *author* field will usually be the server name when joining a channel (on Freenode, at least), and the nick of the user setting the topic when the topic has been changed.

User events

These events occur when a user changes state in some way, i.e. actions that aren't limited to a single channel.

core.user.quit(user, message)

core.user.renamed(oldnick, newnick)

CHAPTER 3

csbot

csbot package

Subpackages

csbot.plugins package

Submodules

csbot.plugins.auth module

class csbot.plugins.auth.**PermissionDB**
Bases: collections.defaultdict

A helper class for assembling the permissions database.

process(*entity*, *permissions*)
Process a configuration entry, where *entity* is an account name, `@group` name or `*` and *permissions* is a space-separated list of permissions to grant.

get_permissions(*entity*)
Get the set of permissions for *entity*.

The union of the permissions for *entity* and the universal (`*`) permissions is returned. If *entity* is `None`, only the universal permissions are returned.

check(*entity*, *permission*, *channel=None*)
Check if *entity* has *permission*.

If *channel* is present, check for a channel permission, otherwise check for a bot permission. Compatible wildcard permissions are also checked.

class csbot.plugins.auth.**Auth**(*bot*)
Bases: *csbot.plugin.Plugin*

PLUGIN_DEPENDS = {'usertrack'}

setup()

check(*nick*, *perm*, *channel=None*)

check_or_error(*e*, *perm*, *channel=None*)

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = []

csbot.plugins.calc module

csbot.plugins.calc.**is_too_long**(*n*)

csbot.plugins.calc.**guarded_power**(*a*, *b*)
A limited power function to make sure that commands do not take too long to process.

csbot.plugins.calc.**guarded_lshift**(*a*, *b*)

csbot.plugins.calc.**guarded_rshift**(*a*, *b*)

csbot.plugins.calc.**guarded_factorial**(*a*)

class csbot.plugins.calc.**CalcEval**
Bases: ast.NodeVisitor

visit_Module(*node*)

visit_Expr(*node*)

visit_BinOp(*node*)

visit_UnaryOp(*node*)

visit_Compare(*node*)

visit_Call(*node*)

visit_Name(*node*)

visit_Num(*node*)

visit_NameConstant(*node*)

visit_Str(*node*)

exception csbot.plugins.calc.**CalcError**
Bases: Exception

class csbot.plugins.calc.**Calc**(*bot*)
Bases: *csbot.plugin.Plugin*

A plugin that calculates things.

do_some_calc(*e*)
What? You don't have a calculator handy?

PLUGIN_DEPENDS = set()

plugin_cmds = [('calc', {'help': 'For calculating, not interpreting'}, <function Calc.do_some_calc>)]

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = []

csbot.plugins.cron module

class csbot.plugins.cron.**Cron**(*bot*)
Bases: *csbot.plugin.Plugin*

Time, that most mysterious of things. What is it? Is it discrete or continuous? What was before time? Does that even make sense to ask? This plugin will attempt to address some, perhaps all, of these questions.

More seriously, this plugin allows the scheduling of events. Due to computers being the constructs of fallible humans, it's not guaranteed that a callback will be run precisely when you want it to be. Furthermore, if you schedule multiple events at the same time, don't make any assumptions about the order in which they'll be called.

Example of usage:

class MyPlugin(Plugin): cron = Plugin.use('cron')

def setup(self): ... self.cron.after(

"hello world", datetime.timedelta(days=1), "callback")

def callback(self, when): self.log.info(u'I got called at {}'.format(when))

@Plugin.hook('cron.hourly') def hourlyevent(self, e):

self.log.info(u'An hour has passed')

tasks = ProvidedByPlugin(plugin='mongodb', kwargs={'collection': 'tasks'})

setup()

teardown()

fire_event(*now*, *name*)
Fire off a regular event.

This gets called by the scheduler at the appropriate time.

provide(*plugin_name*)
Return the crond for the given plugin.

match_task(*owner*, *name=None*, *args=None*, *kwargs=None*)
Create a MongoDB search for a task definition.

schedule(*owner*, *name*, *when*, *interval=None*, *callback=None*, *args=None*, *kwargs=None*)
Schedule a new task.

Parameters

- **owner** – The plugin which created the task
- **name** – The name of the task
- **when** – The datetime to trigger the task at
- **interval** – Optionally, reschedule at when + interval when triggered. Gives rise to repeating tasks.
- **callback** – Call owner.callback when triggered; if None, call owner.name.

- **args** – Callback positional arguments.
- **kwargs** – Callback keyword arguments.

The signature of a task is `(owner, name, args, kwargs)`, and trying to create a task with the same signature as an existing task will raise *DuplicateTaskError*. Any subset of the signature can be used to *unschedule()* all matching tasks (`owner` is mandatory).

unschedule(*owner*, *name=None*, *args=None*, *kwargs=None*)
Unschedule a task.

Removes all existing tasks that match based on the criteria passed as arguments (see *match_task()*).

This could result in the scheduler having nothing to do in its next call, but this isn't a problem as it's not a very intensive function, so there's no point in rescheduling it here.

schedule_event_runner()
Schedule the event runner.

Set up a delayed call for *event_runner()* to happen no sooner than is required by the next scheduled task. If a different call already exists it is replaced.

event_runner()
Run pending tasks.

Run all tasks which have a trigger time in the past, and then reschedule self to run in time for the next task.

PLUGIN_DEPENDS = {'mongodb'}

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = [('tasks', ProvidedByPlugin(plugin='mongodb', kwargs={'collection': 'tasks'}))]

exception csbot.plugins.cron.**DuplicateTaskError**
Bases: Exception

Task with a given signature already exists.

This can be raised by *Cron.schedule()* if a plugin tries to register two events with the same name.

class csbot.plugins.cron.**PluginCron**(*cron*, *plugin*)
Bases: object

Interface to the cron methods restricted to *plugin* as the task owner..

All of the scheduling functions have a signature of the form (name, time, method_name, *args, **kwargs).

This means that at the appropriate time, the method plugin.method_name will be called with the arguments (time, *args, **kwargs), where the time argument is the time it was supposed to be run by the scheduler (which may not be identical to teh actual time it is run).

These functions will raise a DuplicateNameException if you try to schedule two events with the same name.

schedule(*name*, *when*, *interval=None*, *callback=None*, *args=None*, *kwargs=None*)
Pass through to *Cron.schedule()*, adding *owner* argument.

after(*_delay*, *_name*, *_method_name*, **args*, ***kwargs*)
Schedule an event to occur after the timedelta delay has passed.

at(*_when*, *_name*, *_method_name*, **args*, ***kwargs*)
Schedule an event to occur at a given time.

every(*_freq*, *_name*, *_method_name*, **args*, ***kwargs*)
Schedule an event to occur every time the delay passes.

unschedule(*name*, *args=None*, *kwargs=None*)
Pass through to *Cron.unschedule()*, adding *owner* argument.

unschedule_all()
Unschedule all tasks for this plugin.

This could be supported by *unschedule()*, but it's nice to prevent code accidentally wiping all of a plugin's tasks.

csbot.plugins.csyork module

class csbot.plugins.csyork.**CSYork**(*bot*)
Bases: *csbot.plugin.Plugin*

Amusing replacements for various #cs-york members

respond(*e*)

PLUGIN_DEPENDS = set()

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {'core.message.privmsg': [<function CSYork.respond>]})

plugin_integrations = []

plugin_provide = []

csbot.plugins.helix module

class csbot.plugins.helix.**Helix**(*bot*)
Bases: *csbot.plugin.Plugin*

The premier csbot plugin, allowing mere mortals to put questions to the mighty helix, and receive his divine wisdom.

Notes:

- The popular online version basically just selects a random outcome, and saves it with a random url so that it can be reused if the same question is asked.
- I'm lazy, so I'm just going to hash whatever the person puts in and mod the resulting value (taken from hex) to pick out an element of the outcomes list. That way if the same questions gets asked twice, it gets (hopefully) the same answer.

outcomes = ['It is certain', 'It is decidedly so', 'Without a doubt', 'Yes definitely', 'You may rely on it', 'As I see it, yes',

setup()

ask_the_almighty_helix(*e*)
Ask and you shall recieve.

PLUGIN_DEPENDS = set()

plugin_cmds = [('helix', {}, <function Helix.ask_the_almighty_helix>)]

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = []

csbot.plugins.hoogle module

class csbot.plugins.hoogle.**Hoogle**(*bot*)
Bases: *csbot.plugin.Plugin*

CONFIG_DEFAULTS = {'results': 5}

setup()

search_hoogle(*e*)
Search Hoogle with a given string and return the first few (exact number configurable) results.

PLUGIN_DEPENDS = set()

plugin_cmds = [('hoogle', {}, <function Hoogle.search_hoogle>)]

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = []

csbot.plugins.imgur module

class csbot.plugins.imgur.**Imgur**(**args*, ***kwargs*)
Bases: *csbot.plugin.Plugin*

CONFIG_DEFAULTS = {'client_secret': None, 'client_id': None}

CONFIG_ENVVARS = {'client_secret': ['IMGUR_CLIENT_SECRET'], 'client_id': ['IMGUR_CLIENT_ID']}

integrate_with_linkinfo(*linkinfo*)

PLUGIN_DEPENDS = set()

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = [(('linkinfo',), <function Imgur.integrate_with_linkinfo>)]

plugin_provide = []

csbot.plugins.linkinfo module

class csbot.plugins.linkinfo.**LinkInfoHandler**(*filter*, *handler*, *exclusive*)
Bases: tuple

exclusive
Alias for field number 2

filter
Alias for field number 0

handler
Alias for field number 1

class csbot.plugins.linkinfo.**LinkInfoResult**(**args*, ***kwargs*)
Bases: *csbot.util.Struct*

url = Struct.REQUIRED
The URL requested

text = Struct.REQUIRED
Information about the URL

is_error = False
Is an error?

nsfw = False
URL is not safe for work?

is_redundant = False
URL information is redundant? (e.g. duplicated in URL string)

get_message()

class csbot.plugins.linkinfo.**LinkInfo**(**args*, ***kwargs*)
Bases: *csbot.plugin.Plugin*

CONFIG_DEFAULTS = {'minimum_path_match': 0.5, 'minimum_slug_length': 10, 'max_file_ext_length': 6, 'max_respo

register_handler(*filter*, *handler*, *exclusive=False*)
Add a URL handler.

filter should be a function that returns a True-like or False-like value to indicate whether *handler* should be run for a particular URL. The URL is supplied as a `urlparse:ParseResult` instance.

If *handler* is called, it will be as `handler(url, filter(url))`. The filter result is useful for accessing the results of a regular expression filter, for example. The result should be a *LinkInfoResult* instance. If the result is None instead, the processing will fall through to the next handler; this is the best way to signal that a handler doesn't know what to do with a particular URL.

If *exclusive* is True, the fall-through behaviour will not happen, instead terminating the handling with the result of calling *handler*.

register_exclude(*filter*)
Add a URL exclusion filter.

filter should be a function that returns a True-like or False-like value to indicate whether or not a URL should be excluded from the default title-scraping behaviour (after all registered handlers have been tried). The URL is supplied as a `urlparse.ParseResult` instance.

link_command(*e*)
Handle the "link" command.

Fetch information about a specified URL, e.g. `!link http://google.com`. The link can be explicitly marked as NSFW by including the string anywhere in the trailing string, e.g. `!link http://lots-of-porn.com nsfw`.

scan_privmsg(*e*)
Scan the data of PRIVMSG events for URLs and respond with information about them.

get_link_info(*original_url*)
Get information about a URL.

Using the *original_url* string, run the chain of URL handlers and excludes to get a *LinkInfoResult*.

scrape_html_title(*url*)
Scrape the `<title>` tag contents from the HTML page at *url*.

Returns a *LinkInfoResult*.

PLUGIN_DEPENDS = set()

plugin_cmds = [('link', {}, <function LinkInfo.link_command>)]

plugin_hooks = defaultdict(<class 'list'>, {'core.message.privmsg': [<function LinkInfo.scan_privmsg>]})

plugin_integrations = []

plugin_provide = []

csbot.plugins.logger module

class csbot.plugins.logger.**Logger**(*bot*)

Bases: *csbot.plugin.Plugin*

raw_log = <logging.Logger object>

pretty_log = <logging.Logger object>

raw_received(*event*)

raw_sent(*event*)

connected(*event*)

disconnected(*event*)

signedon(*event*)

joined(*event*)

left(*event*)

user_joined(*event*)

user_left(*event*)

names(*event*)

topic(*event*)

privmsg(*event*)

notice(*event*)

action(*event*)

quit(*event*)

renamed(*event*)

command(*event*)

PLUGIN_DEPENDS = set()

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {'core.channel.joined': [<function Logger.user_joined>], 'core.user.quit': [<f

plugin_integrations = []

plugin_provide = []

csbot.plugins.mongodb module

class csbot.plugins.mongodb.**MongoDB**(**args*, ***kwargs*)
Bases: *csbot.plugin.Plugin*

A plugin that provides access to a MongoDB server via pymongo.

CONFIG_DEFAULTS = {'mode': 'uri', 'uri': 'mongodb://localhost:27017/csbot'}

CONFIG_ENVVARS = {'uri': ['MONGOLAB_URI', 'MONGODB_URI']}

PLUGIN_DEPENDS = set()

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = []

provide(*plugin_name*, *collection*)
Get a MongoDB collection for `{plugin_name}__{collection}`.

csbot.plugins.termdates module

class csbot.plugins.termdates.**TermDates**(*bot*)
Bases: *csbot.plugin.Plugin*

A wonderful plugin allowing old people (graduates) to keep track of the ever-changing calendar.

DATE_FORMAT = '%Y-%m-%d'

db_terms = ProvidedByPlugin(plugin='mongodb', kwargs={'collection': 'terms'})

db_weeks = ProvidedByPlugin(plugin='mongodb', kwargs={'collection': 'weeks'})

setup()

termdates(*e*)

week(*e*)

termdates_set(*e*)

PLUGIN_DEPENDS = {'mongodb'}

plugin_cmds = [('termdates.set', {'help': 'termdates.set <aut> <spr> <sum>: set the term dates'}, <function TermDates

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = [('db_terms', ProvidedByPlugin(plugin='mongodb', kwargs={'collection': 'terms'})), ('db_weeks',

csbot.plugins.topic module

class csbot.plugins.topic.**Topic**(*bot*)
Bases: *csbot.plugin.Plugin*

PLUGIN_DEPENDS = {'auth'}

CONFIG_DEFAULTS = {'start': '', 'history': 5, 'end': '', 'sep': '|'}

setup()

config_get(*key*, *channel=None*)
A special implementation of `Plugin.config_get()` which looks at a channel-based configuration subsection before the plugin's configuration section.

topic_changed(*e*)

topic(*e*)

topic_history(*e*)

topic_undo(*e*)

topic_append(*e*)

topic_pop(*e*)

topic_replace(*e*)

topic_insert(*e*)

plugin_cmds = [('topic.push', {'help': 'topic.push <text>: alias of topic.append'}, <function Topic.topic_append>), ('to

plugin_hooks = defaultdict(<class 'list'>, {'core.channel.topic': [<function Topic.topic_changed>]})

plugin_integrations = []

plugin_provide = []

csbot.plugins.usertrack module

class csbot.plugins.usertrack.**UserDict**
Bases: collections.defaultdict

static create_user(*nick*)

copy_or_create(*nick*)

class csbot.plugins.usertrack.**UserTrack**(*bot*)
Bases: *csbot.plugin.Plugin*

setup()

get_user(*nick*)
Get a copy of the user record for *nick*.

account_command(*e*)

PLUGIN_DEPENDS = set()

plugin_cmds = [('account', {'help': 'account [nick]: show Freenode account for a nick, or for yourself if omitted'}, <fun

plugin_hooks = defaultdict(<class 'list'>, {'core.channel.joined': [<function UserTrack._channel_joined>], 'core.user.q

plugin_integrations = []

plugin_provide = []

csbot.plugins.whois module

class csbot.plugins.whois.**Whois**(*bot*)
Bases: *csbot.plugin.Plugin*

Associate data with a user and a channel. Users can update their own data, and it persists over nick changes.

PLUGIN_DEPENDS = {'usertrack', 'mongodb'}

whoisdb = ProvidedByPlugin(plugin='mongodb', kwargs={'collection': 'whois'})

whois_lookup(*nick*, *channel*, *db=None*)
Performs a whois lookup for a nick

whois_set(*nick*, *whois_str*, *channel=None*, *db=None*)

whois_unset(*nick*, *channel=None*, *db=None*)

whois(*e*)
Look up a user by nick, and return what data they have set for themselves (or an error message if there is no data)

setdefault(*e*)

set(*e*)
Allow a user to associate data with themselves for this channel.

unset(*e*)

unsetdefault(*e*)

identify_user(*nick*, *channel=None*)
Identify a user: by account if authed, if not, by nick. Produces a dict suitable for throwing at mongo.

plugin_cmds = [('whois', {'help': 'whois [nick]: show whois data for a nick, or for yourself if omitted'}, <function Who

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = [('whoisdb', ProvidedByPlugin(plugin='mongodb', kwargs={'collection': 'whois'}))]

csbot.plugins.xkcd module

csbot.plugins.xkcd.**fix_json_unicode**(*data*)
Fixes the unicode silliness that is included in the json data. Why Randall, Why?

csbot.plugins.xkcd.**get_info**(*number=None*)
Gets the json data for a particular comic (or the latest, if none provided).

class csbot.plugins.xkcd.**xkcd**(*bot*)
Bases: *csbot.plugin.Plugin*

A plugin that does some xkcd things. Based on williebot xkcd plugin.

exception XKCDError
Bases: Exception

xkcd.**linkinfo_integrate**(*linkinfo*)
Handle recognised xkcd urls.

xkcd.**randall_is_awesome**(*e*)
Well, Randall sucks at unicode actually :(

xkcd.**PLUGIN_DEPENDS = set()**

xkcd.**plugin_cmds = [('xkcd', {}, <function xkcd.randall_is_awesome>)]**

xkcd.**plugin_hooks = defaultdict(<class 'list'>, {})**

xkcd.**plugin_integrations** = [((‘linkinfo’,), <function xkcd.linkinfo_integrate>)]

xkcd.**plugin_provide** = []

csbot.plugins.youtube module

csbot.plugins.youtube.**get_yt_id**(*url*)
Gets the video ID from a urllib ParseResult object.

exception csbot.plugins.youtube.**YoutubeError**(*http_error*)
Bases: Exception

Signifies some error occurred accessing the Youtube API.

This is only used for actual errors, e.g. invalid API key, not failure to find any data matching a query.

Pass the HttpError from the API call as an argument.

class csbot.plugins.youtube.**Youtube**(*bot*)
Bases: *csbot.plugin.Plugin*

A plugin that does some youtube things. Based on williebot youtube plugin.

CONFIG_DEFAULTS = {‘api_key’: ‘’}

CONFIG_ENVVARS = {‘api_key’: [’YOUTUBE_DATA_API_KEY’]}

RESPONSE = ‘”{title}” [{duration}] (by {uploader} at {uploaded}) | Views: {views} [{likes}]’

CMD_RESPONSE = ‘”{title}” [{duration}] (by {uploader} at {uploaded}) | Views: {views} [{likes}] | {link}’

http = None
Hook for mocking HTTP responses to Google API client

setup()

get_video_json(*id*)

linkinfo_integrate(*linkinfo*)
Handle recognised youtube urls.

all_hail_our_google_overlords(*e*)
I for one, welcome our Google overlords.

PLUGIN_DEPENDS = set()

plugin_cmds = [(‘yt’, {}, <function Youtube.all_hail_our_google_overlords>), (‘youtube’, {}, <function Youtube.all_hail

plugin_hooks = defaultdict(<class ‘list’>, {})

plugin_integrations = [((‘linkinfo’,), <function Youtube.linkinfo_integrate>)]

plugin_provide = []

Module contents

Submodules

csbot.core module

exception csbot.core.**PluginError**
Bases: Exception

class csbot.core.**Bot**(*config=None*, *loop=None*)

Bases: *csbot.plugin.SpecialPlugin*, *csbot.irc.IRCClient*

CONFIG_DEFAULTS = {'irc_host': 'irc.freenode.net', 'sourceURL': 'http://github.com/csyork/csbot/', 'auth_method': 'pa

Default configuration values

CONFIG_ENVVARS = {'password': ['IRC_PASS']}

Environment variable fallbacks

available_plugins = {'logger': <class 'csbot.plugins.logger.Logger'>, 'mongodb': <class 'csbot.plugins.mongodb.Mo

Dictionary containing available plugins for loading, using straight.plugin to discover plugin classes under a namespace.

bot_setup()

Load plugins defined in configuration and run setup methods.

bot_teardown()

Run plugin teardown methods.

post_event(*event*)

register_command(*cmd*, *metadata*, *f*, *tag=None*)

unregister_command(*cmd*, *tag=None*)

unregister_commands(*tag*)

signedOn(*event*)

privmsg(*event*)

Handle commands inside PRIVMSGs.

fire_command(*event*)

Dispatch a command event to its callback.

show_commands(*e*)

show_plugins(*e*)

emit_new(*event_type*, *data=None*)

Shorthand for firing a new event.

emit(*event*)

Shorthand for firing an existing event.

connection_made()

connection_lost(*exc*)

send_line(*line*)

line_received(*line*)

on_welcome()

on_joined(*channel*)

on_left(*channel*)

on_privmsg(*user*, *channel*, *message*)

on_notice(*user*, *channel*, *message*)

on_action(*user*, *channel*, *message*)

on_user_joined(*user*, *channel*)

on_user_left(*user*, *channel*, *message*)

on_user_quit(*user*, *message*)

on_user_renamed(*oldnick*, *newnick*)

on_topic_changed(*user*, *channel*, *topic*)

irc_RPL_NAMREPLY(*msg*)

irc_RPL_ENDOFNAMES(*msg*)

on_names(*channel*, *names*, *raw_names*)
Called when the NAMES list for a channel has been received.

on_capabilities_available(*capabilities*)

identify(*target*)
Find the account for a user or all users in a channel.

irc_354(*msg*)
Handle "formatted WHO" responses.

on_user_identified(*user*, *account*)

irc_ACCOUNT(*msg*)
Account change notification from `account-notify` capability.

PLUGIN_DEPENDS = set()

irc_JOIN(*msg*)
Re-implement `JOIN` handler to account for `extended-join` info.

plugin_cmds = [('help', {'help': 'help [command]: show help for command, or show available commands'}, <function H

plugin_hooks = defaultdict(<class 'list'>, {'core.message.privmsg': [<function Bot.privmsg>], 'core.self.connected': [<

plugin_integrations = []

plugin_provide = []

reply(*to*, *message*)
Reply to a nick/channel.

This is not implemented because it should be replaced in the constructor with a reference to a real method, e.g. `self.reply = self.msg`.

csbot.events module

class csbot.events.**ImmediateEventRunner**(*handle_event*)
Bases: object

A very simple blocking event runner for immediate chains of events.

This class is only responsible for making sure chains of events get handled before the next root event happens. The *handle_event* method should be a callable that expects a single argument - it will receive whatever is passed to *post_event()*.

The context manager technique is used to ensure the event runner is left in a usable state if an exception propagates out of it in response to running an event.

post_event(*event*)
Post *event* to be handled soon.

If this is a root event, i.e. this method hasn't been called while handling another event, then the event queue will run immediately and block until the event and all child events have been handled.

If this is a child event, i.e. this method has been called from another event handler, then it will be added to the queue and will be processed before the *post_event()* for the root event exits.

If a chain of events forms a tree, the handling order is equivalent to a breadth-first traversal of the event tree.

class csbot.events.**AsyncEventRunner**(*handle_event*, *loop=None*)
Bases: object

An asynchronous event runner.

Runs on *loop*, and events are passed to *handle_event* which should return an iterable of awaitables (coroutine objects, tasks, etc.).

Parameters

- **handle_event** – Function to turn an event into awaitables
- **loop** – asyncio event loop to use (default: None, use current loop)

post_event(*event*)
Post *event* to be handled soon.

All tasks resulting from calling handle_event() on *event* are created and a future that completes only when those tasks are complete is returned.

The returned future may depend on more than just the tasks resulting from *event*, because new tasks are aggregated into an existing future if there are still outstanding tasks.

class csbot.events.**Event**(*bot*, *event_type*, *data=None*)
Bases: dict

IRC event information.

Events are dicts of event information, plus some attributes which are applicable for all events.

bot = None
The *Bot* which triggered the event.

event_type = None
The name of the event.

datetime = None
The value of datetime.datetime.now() when the event was triggered.

classmethod extend(*event*, *event_type=None*, *data=None*)
Create a new event by extending an existing event.

The main purpose of this classmethod is to duplicate an event as a new event type, preserving existing information. For example:

reply(*message*)
Send a reply.

For messages that have a reply_to key, instruct the *bot* to send a reply.

class csbot.events.**CommandEvent**(*bot*, *event_type*, *data=None*)
Bases: *csbot.events.Event*

classmethod parse_command(*event*, *prefix*, *nick*)
Attempt to create a *CommandEvent* from a core.message.privmsg event.

A command is signified by *event["message"]* starting with the command prefix string followed by one or more non-space characters.

Returns None if *event['message']* wasn't recognised as being a command.

arguments()
Parse *self["data"]* into a list of arguments using *parse_arguments()*. This might raise a ValueError if the string cannot be parsed, e.g. if there are unmatched quotes.

csbot.irc module

exception csbot.irc.**IRCParseError**
Bases: Exception

Raised by *IRCMessage.parse()* when a message can't be parsed.

class csbot.irc.**IRCMessage**
Bases: csbot.irc._IRCMessage

Represents an IRC message.

The IRC message format, paraphrased and simplified from RFC2812, is:

```
message = [":" prefix " "] command {" " parameter} [" :" trailing]
```

This is represented as a namedtuple with the following attributes:

Parameters

- **prefix** (*str or None*) – Prefix part of the message, usually the origin
- **command** (*str*) – IRC command
- **params** (*list of str*) – List of command parameters (including trailing)
- **command_name** (*str*) – Name of IRC command (see below)
- **raw** (*str*) – The raw IRC message

The *command_name* attribute is intended to be the "readable" form of the *command*. Usually it will be the same as *command*, but numeric replies recognised in RFC2812 will have their corresponding name instead.

REGEX = re.compile('(:(?P<prefix>\\S+))?(?P<command>\\S+)(?P<params>((?!:)\\S+)*)(:(?P<trailing>.*))?')
Regular expression to extract message components from a message.

FORCE_TRAILING = {'USER', 'PRIVMSG', 'QUIT'}
Commands to force trailing parameter (:blah) for

classmethod parse(*line*)
Create an *IRCMessage* object by parsing a raw message.

classmethod create(*command*, *params=None*, *prefix=None*)
Create an *IRCMessage* from its core components.

The *raw* and *command_name* attributes will be generated based on the message details.

pretty
Get a more readable version of the raw IRC message.

Pretty much identical to the raw IRC message, but numeric commands that have names end up being NUMERIC/NAME.

pad_params(*length*, *default=None*)
Pad parameters to *length* with *default*.

Useful when a command has optional parameters:

```
>>> msg = IRCMessage.parse(':nick!user@host KICK #channel other')
>>> channel, nick, reason = msg.params
Traceback (most recent call last):
  ...
ValueError: need more than 2 values to unpack
>>> channel, nick, reason = msg.pad_params(3)
```

class csbot.irc.**IRCUser**
Bases: csbot.irc._IRCUser

Provide access to the parts of an IRC user string.

The following parts of the user string are available, set to *None* if that part of the string is absent:

Parameters

- **raw** – Raw user string
- **nick** – Nick of the user
- **user** – Username of the user (excluding leading ~)
- **host** – Hostname of the user

```
>>> u = IRCUser.parse('my_nick!some_user@host.name')
>>> u.nick
'my_nick'
>>> u.user
'some_user'
>>> u.host
'host.name'
```

REGEX = re.compile('(?P<raw>(?P<nick>[^!]+)(!~*(?P<user>[^@]+))?(@(?P<host>.+))?)')
Username parsing regex. Stripping out the "~" might be a Freenode peculiarity...

classmethod parse(*raw*)
Create an *IRCUser* from a raw user string.

class csbot.irc.**IRCCodec**
Bases: codecs.Codec

The encoding scheme to use for IRC messages.

IRC messages are "just bytes" with no encoding made explicit in the protocol definition or the messages. Ideally we'd like to handle IRC messages as proper strings.

encode(*input*, *errors='strict'*)
Encode a message as UTF-8.

decode(*input*, *errors='strict'*)
Decode a message.

IRC messages could pretty much be in any encoding. Here we just try the two most likely candidates: UTF-8, falling back to CP1252. Unfortunately, any encoding where every byte is valid (e.g. CP1252) makes it impossible to detect encoding errors - if *input* isn't UTF-8 or CP1252-compatible, the result might be a bit odd.

class csbot.irc.**IRCClient**(*, *loop=None*, ***kwargs*)
Bases: object

Internet Relay Chat client protocol.

A line-oriented protocol for communicating with IRC servers. It handles receiving data at several layers of abstraction:

- *line_received()*: decoded line
- *message_received()*: parsed *IRCMessage*
- irc_<COMMAND>(msg): called when msg.command == '<COMMAND>'
- on_<event>(...): specific events with specific arguments, e.g. on_quit(user, message)

It also handles sending data at several layers of abstraction:

- *send_line()*: raw IRC command, e.g. self.send_line('JOIN #cs-york-dev')
- *send()*: *IRCMessage*, e.g. self.send(IRCMessage.create('JOIN', params=['#cs-york-dev']))
- <action>(...): e.g. self.join('#cs-york-dev').

The API and implementation is inspired by irc3 and Twisted.

- TODO: limit send rate
- TODO: limit PRIVMSG/NOTICE send length
- TODO: NAMES
- TODO: MODE
- TODO: More sophisticated CTCP? (see Twisted)
- TODO: MOTD?
- TODO: SSL

codec = <csbot.irc.IRCCodec object>
Codec for encoding/decoding IRC messages.

static **DEFAULTS**()
Generate a default configuration. Easier to call this and update the result than relying on dict.copy().

available_capabilities = None
Available client capabilities

enabled_capabilities = None
Enabled client capabilities

run(*run_once=False*)
Run the bot, reconnecting when the connection is lost.

connect()
Connect to the IRC server.

disconnect()
Disconnect from the IRC server.

Use *quit()* for a more graceful disconnect.

read_loop()
Read and dispatch lines until the connection closes.

connection_made()
Callback for successful connection.

Register with the IRC server.

connection_lost(*exc*)
Handle a broken connection by attempting to reconnect.

Won't reconnect if the broken connection was deliberate (i.e. `close()` was called).

line_received(*line*)
Callback for received raw IRC message.

message_received(*msg*)
Callback for received parsed IRC message.

send_line(*data*)
Send a raw IRC message to the server.

Encodes, terminates and sends *data* to the server.

send(*msg*)
Send an *IRCMessage*.

enable_capability(*name*)
Enable client capability *name*.

Should wait for *on_capability_enabled()* before assuming it is enabled.

disable_capability(*name*)
Disable client capability *name*.

Should wait for *on_capability_disabled()* befor assuming it is disabled.

set_nick(*nick*)
Ask the server to set our nick.

join(*channel*)
Join a channel.

leave(*channel*, *message=None*)
Leave a channel, with an optional message.

quit(*message=None*, *reconnect=False*)
Leave the server.

If *reconnect* is False, then the client will not attempt to reconnect after the server closes the connection.

msg(*to*, *message*)
Send *message* to a channel/nick.

act(*to*, *action*)
Send *action* as a CTCP ACTION to a channel/nick.

notice(*to*, *message*)
Send *message* as a NOTICE to a channel/nick.

set_topic(*channel*, *topic*)
Try and set a channel's topic.

get_topic(*channel*)
Ask server to send the topic for *channel*.

Will cause *on_topic_changed()* at some point in the future.

ctcp_query(*to*, *command*, *data=None*)
Send CTCP query.

ctcp_reply(*to*, *command*, *data=None*)
Send CTCP reply.

irc_RPL_WELCOME(*msg*)
Received welcome from server, now we can start communicating.

Welcome should include the accepted nick as the first parameter. This may be different to the nick we requested (e.g. truncated to a maximum length); if this is the case we store the new nick and fire the *on_nick_changed()* event.

irc_ERR_NICKNAMEINUSE(*msg*)
Attempted nick is in use, try another.

Adds an underscore to the end of the current nick. If the server truncated the nick, replaces the last non-underscore with an underscore.

irc_PING(*msg*)
IRC PING/PONG keepalive.

irc_CAP(*msg*)
Dispatch `CAP` subcommands to their own methods.

irc_CAP_LS(*msg*)
Response to `CAP LS`, giving list of available capabilities.

irc_CAP_ACK(*msg*)
Response to `CAP REQ`, acknowledging capability changes.

irc_CAP_NAK(*msg*)
Response to `CAP REQ`, rejecting capability changes.

irc_NICK(*msg*)
Somebody's nick changed.

irc_JOIN(*msg*)
Somebody joined a channel.

irc_PART(*msg*)
Somebody left a channel.

irc_KICK(*msg*)
Somebody was kicked from a channel.

irc_QUIT(*msg*)
Somebody quit the server.

irc_TOPIC(*msg*)
A channel's topic changed.

irc_RPL_TOPIC(*msg*)
Topic notification, usually after joining a channel.

irc_PRIVMSG(*msg*)
Received a `PRIVMSG`.

TODO: Implement CTCP queries.

irc_NOTICE(*msg*)
Received a `NOTICE`.

TODO: Implement CTCP replies.

on_capabilities_available(*capabilities*)
Client capabilities are available.

Called with a set of client capability names when we get a response to `CAP LS`.

on_capability_enabled(*name*)
Client capability enabled.

Called when enabling client capability *name* has been acknowledged.

on_capability_disabled(*name*)
Client capability disabled.

Called when disabling client capability *name* has been acknowledged.

on_welcome()
Successfully signed on to the server.

on_nick_changed(*nick*)
Changed nick.

on_joined(*channel*)
Joined a channel.

on_left(*channel*)
Left a channel.

on_kicked(*channel*, *by*, *reason*)
Kicked from a channel.

on_privmsg(*user*, *to*, *message*)
Received a message, either directly or in a channel.

on_notice(*user*, *to*, *message*)
Received a notice, either directly or in a channel.

on_action(*user*, *to*, *action*)
Received CTCP ACTION. Common enough to deserve its own event.

on_ctcp_query_ACTION(*user*, *to*, *data*)
Turn CTCP ACTION into *on_action()* event.

on_user_renamed(*oldnick*, *newnick*)
User changed nick.

on_user_joined(*user*, *channel*)
User joined a channel.

on_user_left(*user*, *channel*, *message*)
User left a channel.

on_user_kicked(*user*, *channel*, *by*, *reason*)
User kicked from a channel.

on_user_quit(*user*, *message*)
User disconnected.

on_topic_changed(*user*, *channel*, *topic*)
user changed the topic of *channel* to *topic*.

csbot.irc.**main**()

csbot.plugin module

csbot.plugin.**build_plugin_dict**(*plugins*)
Build a dictionary mapping the value of *plugin_name()* to each plugin class in *plugins*. *PluginDuplicate* is raised if more than one plugin has the same name.

exception csbot.plugin.**PluginDuplicate**
Bases: Exception

exception csbot.plugin.**PluginDependencyUnmet**
Bases: Exception

exception csbot.plugin.**PluginFeatureError**
Bases: Exception

class csbot.plugin.**PluginManager**(*loaded*, *available*, *plugins*, *args*)
Bases: collections.abc.Mapping

A simple plugin manager and proxy.

The plugin manager is responsible for loading plugins and proxying method calls to all plugins. In addition to accepting *loaded*, a list of existing plugin objects, it will attempt to load each of *plugins* from *available* (a mapping of plugin name to plugin class), passing *args* to the constructors.

Attempting to load missing or duplicate plugins will log errors and warnings respectively, but will not result in an exception or any change of state. A plugin class' dependencies are checked before loading and a *PluginDependencyUnmet* is raised if any are missing.

The Mapping interface is implemented to provide easy querying and access to the loaded plugins. All attributes that do not start with a _ are treated as methods that will be proxied through to every plugin in the order they were loaded (*loaded* before *plugins*) with the same arguments.

plugins = {}
Loaded plugins.

class csbot.plugin.**ProvidedByPlugin**(*plugin*, *kwargs*)
Bases: tuple

kwargs
Alias for field number 1

plugin
Alias for field number 0

class csbot.plugin.**PluginMeta**(*name*, *bases*, *dict*)
Bases: type

Metaclass for *Plugin* that collects methods tagged with plugin feature decorators.

class csbot.plugin.**Plugin**(*bot*)
Bases: object

Bot plugin base class.

All bot plugins should inherit from this class. It provides convenience methods for hooking events, registering commands, accessing MongoDB and manipulating the configuration file.

CONFIG_DEFAULTS = {}
Default configuration values, used automatically by *config_get()*.

CONFIG_ENVVARS = {}
Configuration environment variables, used automatically by *config_get()*.

PLUGIN_DEPENDS = set()
Plugins that *missing_dependencies()* should check for.

log = None
The plugin's logger, created by default using the plugin class' containing module name as the logger name.

classmethod plugin_name()
Get the name of the plugin, by default the class name in lowercase.

classmethod qualified_name()
Get the fully qualified class name, most useful when complaining about duplicate plugins names.

classmethod missing_dependencies(*plugins*)
Return elements from *PLUGIN_DEPENDS* that are not in the container *plugins*.

This should be used with some container of already loaded plugin names (e.g. a dictionary or set) to find out which dependencies are missing.

static hook(*hook*)

static command(*cmd*, ***metadata*)
Tag a command to be registered by *setup()*.

Additional keyword arguments are added to a metadata dictionary that gets stored with the command. This is a good place to put, for example, the help string for the command:

```
@Plugin.command('foo', help='foo: does something amazing')
def foo_command(self, e):
    pass
```

static integrate_with(**otherplugins*)
Tag a method as providing integration with *otherplugins*.

During *setup()*, all methods tagged with this decorator will be run if all of the named plugins are loaded. The actual plugin objects will be passed as arguments to the method in the same order.

Note: The order that integration methods are called in cannot be guaranteed, because attribute order is not preserved during class creation.

---

static use(*other*, ***kwargs*)
Create a property that will be provided by another plugin.

Returns a *ProvidedByPlugin* instance. *PluginMeta* will collect attributes of this type, and add *other* as an implicit plugin dependency. *setup()* will replace it with a value acquired from the plugin named by *other*. For example:

```
class Foo(Plugin):
    stuff = Plugin.use('mongodb', collection='stuff')
```

will cause *setup()* to replace the `stuff` attribute with:

```
self.bot.plugins[other].provide(self.plugin_name(), **kwargs)
```

fire_hooks(*event*)
Execute all of this plugin's handlers for *event*.

All handlers are treated as coroutine functions, and the return value is a list of all the invoked coroutines.

provide(*plugin_name*, ***kwarg*)
Provide a value for a *Plugin.use()* usage.

setup()
Plugin setup.

- Replace all *ProvidedByPlugin* attributes.
- Fire all plugin integration methods.

•Register all commands provided by the plugin.

teardown()
Plugin teardown.

•Unregister all commands provided by the plugin.

config
Get the configuration section for this plugin.

Uses the `[plugin_name]` section of the configuration file, creating an empty section if it doesn't exist.

See also:

`configparser`

subconfig(*subsection*)
Get a configuration subsection for this plugin.

Uses the `[plugin_name/subsection]` section of the configuration file, creating an empty section if it doesn't exist.

config_get(*key*)
Convenience wrapper proxying `get()` on *config*.

Given a key, this method tries the following in order:

```
self.config[key]
for v in self.CONFIG_ENVVARS[key]:
    os.environ[v]
self.CONFIG_DEFAULTS[key]
```

`KeyError` is raised if none of the methods succeed.

config_getboolean(*key*)
Identical to *config_get()*, but proxying `getboolean`.

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = []

class csbot.plugin.**SpecialPlugin**(*bot*)
Bases: *csbot.plugin.Plugin*

A special plugin with a special name that expects to be handled specially. Probably shouldn't have too many of these or they won't feel special anymore.

PLUGIN_DEPENDS = set()

plugin_cmds = []

plugin_hooks = defaultdict(<class 'list'>, {})

plugin_integrations = []

plugin_provide = []

classmethod plugin_name()
Change the plugin name to something that can't possibly result from a class name by prepending a `@`.

csbot.util module

class csbot.util.**User**(*raw*)
Bases: object

csbot.util.**nick**(*user*)
Get nick from user string.

```
>>> nick('csyorkbot!~csbot@example.com')
'csyorkbot'
```

csbot.util.**username**(*user*)
Get username from user string.

```
>>> username('csyorkbot!~csbot@example.com')
'csbot'
```

csbot.util.**host**(*user*)
Get hostname from user string.

```
>>> host('csyorkbot!~csbot@example.com')
'example.com'
```

csbot.util.**is_channel**(*channel*)
Check if *channel* is a channel or private chat.

```
>>> is_channel('#cs-york')
True
>>> is_channel('csyorkbot')
False
```

csbot.util.**parse_arguments**(*raw*)
Parse *raw* into a list of arguments using shlex.

The shlex lexer is customised to be more appropriate for grouping natural language arguments by only treating " as a quote character. This allows ' to be used naturally. A ValueError will be raised if the string couldn't be parsed.

```
>>> parse_arguments("a test string")
['a', 'test', 'string']
>>> parse_arguments("apostrophes aren't a problem")
['apostrophes', "aren't", 'a', 'problem']
>>> parse_arguments('"string grouping" is useful')
['string grouping', 'is', 'useful']
>>> parse_arguments('just remember to "match your quotes')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: No closing quotation
```

csbot.util.**simple_http_get**(*url*, *stream=False*)
A deliberately dumb wrapper around requests.get().

This should be used for the vast majority of HTTP GET requests. It turns off SSL certificate verification and sets a non-default User-Agent, thereby succeeding at most "just get the content" requests.

stream controls the "streaming mode" of the HTTP client, i.e. deferring the acquisition of the response body. Use this if you need to impose a maximum size or process a large response. *The entire content must be consumed or ''response.close()'' must be called.*

csbot.util.**pairwise**(*iterable*)
Pairs elements of an iterable together, e.g. s -> (s0,s1), (s1,s2), (s2, s3), ...

csbot.util.**cap_string**(*s*, *l*)
If a string is longer than a particular length, it gets truncated and has '...' added to the end.

csbot.util.**ordinal**(*value*)
Converts zero or a *postive* integer (or their string representations) to an ordinal value.

http://code.activestate.com/recipes/576888-format-a-number-as-an-ordinal/

```
>>> for i in range(1,13):
...     ordinal(i)
...
u'1st'
u'2nd'
u'3rd'
u'4th'
u'5th'
u'6th'
u'7th'
u'8th'
u'9th'
u'10th'
u'11th'
u'12th'
```

```
>>> for i in (100, '111', '112',1011):
...     ordinal(i)
...
u'100th'
u'111th'
u'112th'
u'1011th'
```

csbot.util.**pluralize**(*n*, *singular*, *plural*)

csbot.util.**is_ascii**(*s*)
Returns true if all characters in a string can be represented in ASCII.

class csbot.util.**NamedObject**(*name*)
Bases: object

Make objects that have specific __repr__() text.

This is mostly useful for singleton objects that you want to give a useful description for auto-generated documentation purposes.

class csbot.util.**StructMeta**
Bases: type

A metaclass for *Struct* to turn class attributes into fields.

class csbot.util.**Struct**(**args*, ***kwargs*)
Bases: object

A mutable alternative to collections.namedtuple().

To use this class, create a subclass of it. Any non-callable, non-"hidden" class attributes in the subclass will become struct fields. Setting of attribute values is limited to attributes recognised as fields. The class attribute value is effectively the field's default value.

A struct constructor allows both positional arguments (based on field order) and keyword arguments (based on field name). If a field's default value is *REQUIRED*, then an exception will be raised unless its value was set by the constructor.

Examples:

```
>>> class Foo(Struct):
...     a = Struct.REQUIRED
...     b = 12
...     c = None
...
>>> Foo()
Traceback (most recent call last):
    ...
ValueError: value required for attribute: a
>>> Foo(123)
Foo(a=123, b=12, c=None)
>>> Foo(123, c='Hello, world')
Foo(a=123, b=12, c='Hello, world')
>>> Foo(bar=False)
Traceback (most recent call last):
    ...
AttributeError: struct field does not exist: bar
>>> x = Foo(123)
>>> x.b
12
>>> x.b = 21
>>> x
Foo(a=123, b=21, c=None)
>>> x.bar = False
Traceback (most recent call last):
    ...
AttributeError: struct field does not exist: bar
```

REQUIRED = Struct.REQUIRED
Singleton object to signify an attribute that *must* be set

Module contents

class csbot.**PrettyStreamHandler**(*stream=None*, *colour=None*)
Bases: logging.StreamHandler

Wrap log messages with severity-dependent ANSI terminal colours.

Use in place of logging.StreamHandler to have log messages coloured according to severity.

```
>>> handler = PrettyStreamHandler()
>>> handler.setFormatter(logging.Formatter('[%(levelname)-8s] %(message)s'))
>>> logging.getLogger('').addHandler(handler)
```

stream corresponds to the same argument to logging.StreamHandler, defaulting to stderr.

colour overrides TTY detection to force colour on or off.

This source for this class is released into the public domain.

Code author: Alan Briolat <alan.briolat@gmail.com>

COLOURS = {40: '\x1b[31m', 10: '\x1b[36m', 50: '\x1b[31;7m', 30: '\x1b[33m'}
Mapping from logging levels to ANSI colours.

COLOUR_END = '\x1b[0m'
ANSI code for resetting the terminal to default colour.

format(*record*)
Get a coloured, formatted message for a log record.

Calls `logging.StreamHandler.format()` and applies a colour to the message if appropriate.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y