SoCo (Sonos Controller) Documentation

Release 0.11

Rahim Sonawalla, et al.

June 08, 2015

Contents

SoCo (Sonos Controller) is a Python library to control your Sonos speakers.

CHAPTER 1

Contents

1.1 Tutorial

SoCo allows you to control your Sonos sound system from a Python program. For a quick start have a look at the example applications that come with the library.

1.1.1 Discovery

For discovering the Sonos devices in your network, use the `soco.discover()` method.

```
zones = list(soco.discover())
```

1.1.2 Music

Once one of the available devices is selected, the `SoCo` class can be used to control it. Have a look at the *The soco module* for all available commands.

```
sonos = SoCo(ip)
sonos.partymode()
```

1.2 The soco module

SoCo (Sonos Controller) is a simple library to control Sonos speakers

soco.**discover**(*timeout=1*, *include_invisible=False*, *interface_addr=None*)

Discover Sonos zones on the local network.

Return an set of SoCo instances for each zone found. Include invisible zones (bridges and slave zones in stereo pairs if *include_invisible* is True. Will block for up to *timeout* seconds, after which return *None* if no zones found.

Parameters

- **timeout** (*int*) – block for this many seconds, at most. Default 1
- **include_invisible** (*bool*) – include invisible zones in the return set. Default False

- **interface_addr** (*str*) – Discovery operates by sending UDP multicast datagrams. interface_addr is a string (dotted quad) representation of the network interface address to use as the source of the datagrams (i.e. it is a value for IP_MULTICAST_IF). If None or not specified, the system default interface for UDP multicast messages will be used. This is probably what you want to happen.

Returns a set of SoCo instances, one for each zone found, or else None.

Return type (set)

Note: There is no easy cross-platform way to find out the addresses of the local machine's network interfaces. You might try the netifaces module and some code like this:

```
>>> from netifaces import interfaces, AF_INET, ifaddresses
>>> data = [ifaddresses(i) for i in interfaces()]
>>> [d[AF_INET][0]['addr'] for d in data if d.get(AF_INET)]
['127.0.0.1', '192.168.1.20']

This should provide you with a list of values to try for
interface_addr if you are having trouble finding your Sonos devices
```

---

class soco.**SoCo**(*ip_address*)

A simple class for controlling a Sonos speaker.

For any given set of arguments to __init__, only one instance of this class may be created. Subsequent attempts to create an instance with the same arguments will return the previously created instance. This means that all SoCo instances created with the same ip address are in fact the *same* SoCo instance, reflecting the real world position.

Public functions:

```
play -- Plays the current item.
play_uri -- Plays a track or a music stream by URI.
play_from_queue -- Plays an item in the queue.
pause -- Pause the currently playing track.
stop -- Stop the currently playing track.
seek -- Move the currently playing track a given elapsed time.
next -- Go to the next track.
previous -- Go back to the previous track.
switch_to_line_in -- Switch the speaker's input to line-in.
switch_to_tv -- Switch the playbar speaker's input to TV.
get_current_track_info -- Get information about the currently playing
                          track.
get_speaker_info -- Get information about the Sonos speaker.
partymode -- Put all the speakers in the network in the same group.
join -- Join this speaker to another "master" speaker.
unjoin -- Remove this speaker from a group.
get_queue -- Get information about the queue.
get_artists -- Get artists from the music library
get_album_artists -- Get album artists from the music library
get_albums -- Get albums from the music library
get_genres -- Get genres from the music library
get_composers -- Get composers from the music library
get_tracks -- Get tracks from the music library
get_playlists -- Get playlists from the music library
get_music_library_information -- Get information from the music library
get_current_transport_info -- get speakers playing state
browse_by_idstring -- Browse (get sub-elements) a given type
add_uri_to_queue -- Adds an URI to the queue
```

```
add_to_queue -- Add a track to the end of the queue
remove_from_queue -- Remove a track from the queue
clear_queue -- Remove all tracks from queue
get_favorite_radio_shows -- Get favorite radio shows from Sonos'
                            Radio app.
get_favorite_radio_stations -- Get favorite radio stations.
create_sonos_playlist -- Create a new empty Sonos playlist
create_sonos_playlist_from_queue -- Create a new Sonos playlist
                                    from the current queue.
add_item_to_sonos_playlist -- Adds a queueable item to a Sonos'
                              playlist
get_item_album_art_uri -- Get an item's Album Art absolute URI.
search_track -- Search for an artist, artist's albums, or track.
get_albums_for_artist -- Get albums for an artist.
get_tracks_for_album -- Get tracks for an artist's album.
start_library_update -- Trigger an update of the music library.
```

Properties:

```
uid -- The speaker's unique identifier
mute -- The speaker's mute status.
volume -- The speaker's volume.
bass -- The speaker's bass EQ.
treble -- The speaker's treble EQ.
loudness -- The status of the speaker's loudness compensation.
cross_fade -- The status of the speaker's crossfade.
status_light -- The state of the Sonos status light.
player_name  -- The speaker's name.
play_mode -- The queue's repeat/shuffle settings.
queue_size -- Get size of queue.
library_updating -- Whether music library update is in progress.
album_artist_display_option -- album artist display option
is_playing_tv -- Is the playbar speaker input from TV?
is_playing_radio -- Is the speaker input from radio?
is_playing_line_in -- Is the speaker input from line-in?
```

Warning: These properties are not cached and will obtain information over the network, so may take longer than expected to set or return a value. It may be a good idea for you to cache the value in your own code.

add_item_to_sonos_playlist(*queueable_item*, *sonos_playlist*)
Adds a queueable item to a Sonos' playlist

Parameters

- **queueable_item** – the item to add to the Sonos' playlist
- **sonos_playlist** – the Sonos' playlist to which the item should be added

add_to_queue(**args*, ***kwargs*)
Adds a queueable item to the queue

add_uri_to_queue(**args*, ***kwargs*)
Adds the URI to the queue

Parameters **uri** (*str*) – The URI to be added to the queue

album_artist_display_option
Return the current value of the album artist compilation setting (see http://www.sonos.com/support/help/3.4/en/sonos_user_guide/ Chap07_new/Compilation_albums.htm)

This is a string. Possible values:

- "WMP" - Use Album Artists
- "ITUNES" - Use iTunes® Compilations
- "NONE" - Do not group compilations

To change the current setting, call *start_library_update* and pass the new setting.

all_groups
Return a set of all the available groups

all_zones
Return a set of all the available zones

bass
The speaker's bass EQ. An integer between -10 and 10.

browse(*ml_item=None*, *start=0*, *max_items=100*, *full_album_art_uri=False*, *search_term=None*, *subcategories=None*)
Browse (get sub-elements) a music library item

Parameters

- **ml_item** (*MusicLibraryItem*) – The MusicLibraryItem to browse, if left out or passed None, the items at the base level will be returned
- **start** (*int*) – The starting index of the results
- **max_items** (*int*) – The maximum number of items to return
- **full_album_art_uri** (*bool*) – If the album art URI should include the IP address
- **search_term** (*str*) – A string that will be used to perform a fuzzy search among the search results. If used in combination with subcategories, the fuzzy search will be performed on the subcategory. NOTE: Searching will not work if ml_item is None.
- **subcategories** (*list*) – A list of strings that indicate one or more subcategories to dive into. NOTE: Providing sub categories will not work if ml_item is None.

Returns A `SearchResult` object

Return type `SearchResult`

Raises AttributeError: If `ml_item` has no `item_id` attribute SoCoUPnPException: With `error_code='701'` if the item cannot be browsed

browse_by_idstring(*search_type*, *idstring*, *start=0*, *max_items=100*, *full_album_art_uri=False*)
Browse (get sub-elements) a given type

Parameters

- **search_type** – The kind of information to retrieve. Can be one of: 'artists', 'album_artists', 'albums', 'genres', 'composers', 'tracks', 'share', 'sonos_playlists', and 'playlists', where playlists are the imported file based playlists from the music library
- **idstring** – String ID to search for
- **start** – Starting number of returned matches
- **max_items** – Maximum number of returned matches. NOTE: The maximum may be restricted by the unit, presumably due to transfer size consideration, so check the returned number against the requested.
- **full_album_art_uri** – If the album art URI should include the IP address

Returns A dictionary with metadata for the search, with the keys 'number_returned', 'update_id', 'total_matches' and an 'item_list' list with the search results.

clear_queue(**args*, ***kwargs*)
Removes all tracks from the queue.

Returns: True if the Sonos speaker cleared the queue.

Raises SoCoException (or a subclass) upon errors.

create_sonos_playlist(*title*)
Create a new empty Sonos playlist.

Params title Name of the playlist

Returns An instance of *DidlPlaylistContainer*

create_sonos_playlist_from_queue(**args*, ***kwargs*)
Create a new Sonos playlist from the current queue.

Params title Name of the playlist

Returns An instance of *DidlPlaylistContainer*

cross_fade
The speaker's cross fade state. True if enabled, False otherwise

get_album_artists(**args*, ***kwargs*)
Convenience method for *get_music_library_information()* with *search_type='album_artists'*. For details on remaining arguments refer to the docstring for that method.

get_albums(**args*, ***kwargs*)
Convenience method for *get_music_library_information()* with *search_type='albums'*. For details on remaining arguments refer to the docstring for that method.

get_albums_for_artist(*artist*, *full_album_art_uri=False*)
Get albums for an artist.

Parameters

- **artist** (*str*) – Artist name
- **full_album_art_uri** (*bool*) – If the album art URI should include the IP address

Returns A `SearchResult` object.

Return type `SearchResult`

get_artists(**args*, ***kwargs*)
Convenience method for *get_music_library_information()* with *search_type='artists'*. For details on remaining arguments refer to the docstring for that method.

get_composers(**args*, ***kwargs*)
Convenience method for *get_music_library_information()* with *search_type='composers'*. For details on remaining arguments refer to the docstring for that method.

get_current_track_info()
Get information about the currently playing track.

Returns: A dictionary containing the following information about the currently playing track: playlist_position, duration, title, artist, album, position and a link to the album art.

If we're unable to return data for a field, we'll return an empty string. This can happen for all kinds of reasons so be sure to check values. For example, a track may not have complete metadata and be missing an album name. In this case track['album'] will be an empty string.

get_current_transport_info()
Get the current playback state

Returns: A dictionary containing the following information about the speakers playing state current_transport_state (PLAYING, PAUSED_PLAYBACK, STOPPED), current_trasnport_status (OK, ?), current_speed(1,?)

This allows us to know if speaker is playing or not. Don't know other states of CurrentTransportStatus and CurrentSpeed.

get_favorite_radio_shows(*start=0*, *max_items=100*)
Get favorite radio shows from Sonos' Radio app.

Returns: A list containing the total number of favorites, the number of favorites returned, and the actual list of favorite radio shows, represented as a dictionary with *title* and *uri* keys.

Depending on what you're building, you'll want to check to see if the total number of favorites is greater than the amount you requested (*max_items*), if it is, use *start* to page through and get the entire list of favorites.

get_favorite_radio_stations(*start=0*, *max_items=100*)
Get favorite radio stations from Sonos' Radio app.

Returns: A list containing the total number of favorites, the number of favorites returned, and the actual list of favorite radio stations, represented as a dictionary with *title* and *uri* keys.

Depending on what you're building, you'll want to check to see if the total number of favorites is greater than the amount you requested (*max_items*), if it is, use *start* to page through and get the entire list of favorites.

get_genres(**args*, ***kwargs*)
Convenience method for *get_music_library_information()* with *search_type='genres'*. For details on remaining arguments refer to the docstring for that method.

get_item_album_art_uri(*item*)
Get an item's Album Art absolute URI.

get_music_library_information(*search_type*, *start=0*, *max_items=100*, *full_album_art_uri=False*, *search_term=None*, *subcategories=None*, *complete_result=False*)
Retrieve music information objects from the music library

This method is the main method to get music information items, like e.g. tracks, albums etc., from the music library with. It can be used in a few different ways:

The **search_term** argument performs a fuzzy search on that string in the results, so e.g calling:

```
get_music_library_items('artist', search_term='Metallica')
```

will perform a fuzzy search for the term 'Metallica' among all the artists.

Using the **subcategories** argument, will jump directly into that subcategory of the search and return results from there. So. e.g knowing that among the artist is one called 'Metallica', calling:

```
get_music_library_items('artist', subcategories=['Metallica'])
```

will jump directly into the 'Metallica' sub category and return the albums associated with Metallica and:

```
get_music_library_items('artist', subcategories=['Metallica',
                                                 'Black'])
```

will return the tracks of the album 'Black' by the artist 'Metallica'. The order of sub category types is: Genres->Artists->Albums->Tracks. It is also possible to combine the two, to perform a fuzzy search in a sub category.

The **start**, **max_items** and **complete_result** arguments all has to do with paging of the results. Per default, the searches are always paged, because there is a limit to how many items we can get at a time. This paging is exposed to the user with the start and max_items arguments. So calling:

```
get_music_library_items('artists', start=0, max_items=100)
get_music_library_items('artists', start=100, max_items=100)
```

will get the first and next 100 items, respectively. It is also possible to ask for all the elements at once:

```
get_music_library_items('artists', complete_result=True)
```

This will perform the paging internally and simply return all the items.

Parameters

- **search_type** – The kind of information to retrieve. Can be one of: 'artists', 'album_artists', 'albums', 'genres', 'composers', 'tracks', 'share', 'sonos_playlists', and 'playlists', where playlists are the imported file based playlists from the music library
- **start** – Starting number of returned matches (zero based).
- **max_items** – Maximum number of returned matches. NOTE: The maximum may be restricted by the unit, presumably due to transfer size consideration, so check the returned number against the requested.
- **full_album_art_uri** – If the album art URI should include the IP address
- **search_term** – A string that will be used to perform a fuzzy search among the search results. If used in combination with subcategories, the fuzzy search will be performed in the subcategory
- **subcategories** – A list of strings that indicate one or more subcategories to dive into
- **complete_result** – Will disable paging (ignore start and max_items) and return all results for the search. WARNING! Getting e.g. all the tracks in a large collection might take some time.

Returns A `SearchResult` object

Raises *SoCoException* upon errors

NOTE: The playlists that are returned with the 'playlists' search, are the playlists imported from (files in) the music library, they are not the Sonos playlists.

The information about the which searches can be performed and the form of the query has been gathered from the Janos project: http://sourceforge.net/projects/janos/ Props to the authors of that project.

get_playlists(**args*, ***kwargs*)

Convenience method for *get_music_library_information()* with *search_type='playlists'*. For details on remaining arguments refer to the docstring for that method.

NOTE: The playlists that are referred to here are the playlist (files) imported from the music library, they are not the Sonos playlists.

get_queue(*start=0*, *max_items=100*, *full_album_art_uri=False*)

Get information about the queue

Parameters

- **start** – Starting number of returned matches
- **max_items** – Maximum number of returned matches
- **full_album_art_uri** – If the album art URI should include the IP address

Returns A `Queue` object

This method is heavly based on Sam Soffes (aka soffes) ruby implementation

get_sonos_playlists(**args*, ***kwargs*)
Convenience method for: get_music_library_information('sonos_playlists') Refer to the docstring for that method

get_speaker_info(*refresh=False*)
Get information about the Sonos speaker.

Arguments: refresh – Refresh the speaker info cache.

Returns: Information about the Sonos speaker, such as the UID, MAC Address, and Zone Name.

get_tracks(**args*, ***kwargs*)
Convenience method for `get_music_library_information()` with *search_type='tracks'*. For details on remaining arguments refer to the docstring for that method.

get_tracks_for_album(*artist*, *album*, *full_album_art_uri=False*)
Get tracks for an artist's album.

Parameters

- **artist** (*str*) – Artist name
- **album** (*str*) – Album name
- **full_album_art_uri** (*bool*) – If the album art URI should include the IP address

Returns A `SearchResult` object.

Return type `SearchResult`

group
The Zone Group of which this device is a member.

group will be None if this zone is a slave in a stereo pair.

is_bridge
Is this zone a bridge?

is_coordinator
Return True if this zone is a group coordinator, otherwise False.

return True or False

is_playing_line_in
Is the speaker playing line-in?

return True or False

is_playing_radio
Is the speaker playing radio?

return True or False

is_playing_tv
Is the playbar speaker input from TV?

return True or False

is_visible
Is this zone visible? A zone might be invisible if, for example it is a bridge, or the slave part of stereo pair.

return True or False

join(*master*)
Join this speaker to another "master" speaker.

Note: The signature of this method has changed in 0.8. It now requires a SoCo instance to be passed as *master*, not an IP address

---

library_updating
True if the music library is in the process of being updated

Returns True if the music library is in the process of being updated

Return type bool

loudness
The Sonos speaker's loudness compensation. True if on, otherwise False.

Loudness is a complicated topic. You can find a nice summary about this feature here: http://forums.sonos.com/showthread.php?p=4698#post4698

mute
The speaker's mute state. True if muted, False otherwise

next(**args*, ***kwargs*)
Go to the next track.

Returns: True if the Sonos speaker successfully skipped to the next track.

Raises SoCoException (or a subclass) upon errors.

Keep in mind that next() can return errors for a variety of reasons. For example, if the Sonos is streaming Pandora and you call next() several times in quick succession an error code will likely be returned (since Pandora has limits on how many songs can be skipped).

partymode()
Put all the speakers in the network in the same group, a.k.a Party Mode.

This blog shows the initial research responsible for this: http://blog.travelmarx.com/2010/06/exploring-sonos-via-upnp.html

The trick seems to be (only tested on a two-speaker setup) to tell each speaker which to join. There's probably a bit more to it if multiple groups have been defined.

pause(**args*, ***kwargs*)
Pause the currently playing track.

Returns: True if the Sonos speaker successfully paused the track.

Raises SoCoException (or a subclass) upon errors.

play(**args*, ***kwargs*)
Play the currently selected track.

Returns: True if the Sonos speaker successfully started playing the track.

Raises SoCoException (or a subclass) upon errors.

play_from_queue(**args*, ***kwargs*)
Play a track from the queue by index. The index number is required as an argument, where the first index is 0.

index: the index of the track to play; first item in the queue is 0 start: If the item that has been set should start playing

Returns: True if the Sonos speaker successfully started playing the track. False if the track did not start (this may be because it was not requested to start because "start=False")

Raises SoCoException (or a subclass) upon errors.

play_mode
The queue's play mode. Case-insensitive options are:

NORMAL – Turns off shuffle and repeat. REPEAT_ALL – Turns on repeat and turns off shuffle. SHUFFLE – Turns on shuffle *and* repeat. (It's strange, I know.) SHUFFLE_NOREPEAT – Turns on shuffle and turns off repeat.

play_uri(**args*, ***kwargs*)
Play a given stream. Pauses the queue. If there is no metadata passed in and there is a title set then a metadata object will be created. This is often the case if you have a custom stream, it will need at least the title in the metadata in order to play.

Arguments: uri – URI of a stream to be played. meta – The track metadata to show in the player, DIDL format. title – The track title to show in the player start – If the URI that has been set should start playing

Returns: True if the Sonos speaker successfully started playing the track. False if the track did not start (this may be because it was not requested to start because "start=False")

Raises SoCoException (or a subclass) upon errors.

player_name
The speaker's name. A string.

previous(**args*, ***kwargs*)
Go back to the previously played track.

Returns: True if the Sonos speaker successfully went to the previous track.

Raises SoCoException (or a subclass) upon errors.

Keep in mind that previous() can return errors for a variety of reasons. For example, previous() will return an error code (error code 701) if the Sonos is streaming Pandora since you can't go back on tracks.

queue_size
Get size of queue

remove_from_queue(**args*, ***kwargs*)
Remove a track from the queue by index. The index number is required as an argument, where the first index is 0.

index: the index of the track to remove; first item in the queue is 0

Returns True if the Sonos speaker successfully removed the track

Raises SoCoException (or a subclass) upon errors.

search_track(*artist*, *album=None*, *track=None*, *full_album_art_uri=False*)
Search for an artist, artist's albums, or specific track.

Parameters

- **artist** (*str*) – Artist name

- **album** (*str*) – Album name
- **track** (*str*) – Track name
- **full_album_art_uri** (*bool*) – If the album art URI should include the IP address

Returns A `SearchResult` object.

Return type `SearchResult`

seek(**args*, ***kwargs*)
Seeks to a given timestamp in the current track, specified in the format of HH:MM:SS or H:MM:SS.

Returns: True if the Sonos speaker successfully seeked to the timecode.

Raises SoCoException (or a subclass) upon errors.

start_library_update(*album_artist_display_option=u''*)
Start an update of the music library.

If specified, album_artist_display_option changes the album artist compilation setting (see also album_artist_display_option).

status_light
The white Sonos status light between the mute button and the volume up button on the speaker. True if on, otherwise False.

stop(**args*, ***kwargs*)
Stop the currently playing track.

Returns: True if the Sonos speaker successfully stopped the playing track.

Raises SoCoException (or a subclass) upon errors.

switch_to_line_in()
Switch the speaker's input to line-in.

Returns: True if the Sonos speaker successfully switched to line-in.

If an error occurs, we'll attempt to parse the error and return a UPnP error code. If that fails, the raw response sent back from the Sonos speaker will be returned.

Raises SoCoException (or a subclass) upon errors.

switch_to_tv()
Switch the playbar speaker's input to TV.

Returns: True if the Sonos speaker successfully switched to TV.

If an error occurs, we'll attempt to parse the error and return a UPnP error code. If that fails, the raw response sent back from the Sonos speaker will be returned.

Raises SoCoException (or a subclass) upon errors.

treble
The speaker's treble EQ. An integer between -10 and 10.

uid
A unique identifier. Looks like: RINCON_000XXXXXXXXXX1400

unjoin()
Remove this speaker from a group.

Seems to work ok even if you remove what was previously the group master from it's own group. If the speaker was not in a group also returns ok.

Returns: True if this speaker has left the group.

Raises SoCoException (or a subclass) upon errors.

visible_zones
Return an set of all visible zones

volume
The speaker's volume. An integer between 0 and 100.

exception soco.**SoCoException**
base exception raised by SoCo, containing the UPnP error code

exception soco.**UnknownSoCoException**
raised if reason of the error can not be extracted

The exception object will contain the raw response sent back from the speaker

1.3 Plugins

Plugins can extend the functionality of SoCo.

1.3.1 Creating a Plugin

To write a plugin, simply extend the class soco.plugins.SoCoPlugin. The __init__ method of the plugin should accept an SoCo instance as the first positional argument, which it should pass to its super constructor.

The class soco.plugins.example.ExamplePlugin contains an example plugin implementation.

1.3.2 Using a Plugin

To use a plugin, it can be loaded and instantiated directly.

```
# create a plugin by normal instantiation
from soco.plugins.example import ExamplePlugin

# create a new plugin, pass the soco instance to it
myplugin = ExamplePlugin(soco, 'a user')

# do something with your plugin
print 'Testing', myplugin.name
myplugin.music_plugin_stop()
```

Alternatively a plugin can also be loaded by its name using SoCoPlugin.from_name().

```
# get a plugin by name (eg from a config file)
myplugin = SoCoPlugin.from_name('soco.plugins.example.ExamplePlugin',
                                soco, 'some user')

# do something with your plugin
print 'Testing', myplugin.name
myplugin.music_plugin_play()
```

1.3.3 The SoCoPlugin class

class soco.plugins.**SoCoPlugin**(*soco*)
The base class for SoCo plugins

classmethod from_name (*fullname*, *soco*, **args*, ***kwargs*)
Instantiate a plugin by its full name

name
human-readable name of the plugin

1.4 Unit and integration tests

There are two sorts of tests written for the SoCo package. Unit tests implement elementary checks of whether the individual methods produce the expected results. Integration tests check that the package as a whole is able to interface propertly with the Sonos hardware. Such tests are especially useful during re-factoring and to check that already implemented functionality continues to work past updates to the Sonos units' internal software.

1.4.1 Setting up your environment

To run the unit tests, you will need to have the py.test testing tool installed. You will also need a copy of Mock

Mock comes with Python >=3.3, but has been backported for Python 2.7

You can install them and other development dependencies using the *requirements-dev.txt* file like this:

```
pip install -r requirements-dev.txt
```

1.4.2 Running the unit tests

There are different ways of running the unit tests. The easiest is to use py.test's automatic test discovery. Just change to the root directory of the SoCo package and type:

```
py.test
```

For others, see the py.test documentation

1.4.3 Running the integration tests

At the moment, the integration tests cannot be run under the control of py.test. To run them, enter the unittest folder in the source code checkout and run the test execution script execute_unittests.py (it is required that the *SoCo* checkout is added to the Python path of your system). To run all the unit tests for the *SoCo* class execute the following command:

```
python execute_unittests.py --modules soco --ip 192.168.0.110
```

where the IP address should be replaced with the IP address of the Sonos® unit you want to use for the unit tests (NOTE! At present the unit tests for the *SoCo* module requires your Sonos® unit to be playing local network music library tracks from the queue and have at least two such tracks in the queue). You can get a list of all the units in your network and their IP addresses by running:

```
python execute_unittests.py --list
```

To get the help for the unit test execution script which contains a description of all the options run:

```
python execute_unittests.py --help
```

1.4.4 Unit test code structure and naming conventions

The unit tests for the *SoCo* code should be organized according to the following guidelines.

One unit test module per class under test

Unit tests should be organized into modules, one module, i.e. one file, for each class that should be tested. The module should be named similarly to the class except replacing CamelCase with underscores and followed by `_unittest.py`.

Example: Unit tests for the class `FooBar` should be stored in `foo_bar_unittests.py`.

One unit test class per method under test

Inside the unit test modules the unit test should be organized into one unit test case class per method under test. In order for the test execution script to be able to calculate the test coverage, the test classes should be named the same as the methods under test except that the lower case underscores should be converted to CamelCase. If the method is private, i.e. prefixed with 1 or 2 underscores, the test case class name should be prefixed with the word `Private`.

Examples:

Name of method under test	Name of test case class
`get_current_track_info`	`GetCurrentTrackInfo`
`__parse_error`	`PrivateParseError`
`_my_hidden_method`	`PrivateMyHiddenMethod`

1.4.5 Add an unit test to an existing unit test module

To add a unit test case to an existing unit test module `Foo` first check with the following command which methods that does not yet have unit tests:

```
python execute_unittests.py --modules foo --coverage
```

After having identified a method to write a unit test for, consider what criteria should be tested, e.g. if the method executes and returns the expected output on valid input and if it fails as expected on invalid input. Then implement the unit test by writing a class for it, following the naming convention mentioned in section *One unit test class per method under test*. You can read more about unit test classes in the reference documentation and there is a good introduction to unit testing in Mark Pilgrim's "Dive into Python" (though the aspects of test driven development, that it describes, is not a requirement for *SoCo* development).

Special unit test design consideration for *SoCo*

SoCo is developed purely by volunteers in their spare time. This leads to some special consideration during unit test design.

First of, volunteers will usually not have extra Sonos® units dedicated for testing. For this reason the unit tests should be developed in such a way that they can be run on units in use and with people around, so e.g it should be avoided settings the volume to max.

Second, being developed in peoples spare time, the development is likely a recreational activity, that might just be accompanied by music from the same unit that should be tested. For this reason, that unit should be left in the same state after test as it was before. That means that the play list, play state, sound settings etc. should be restored after the testing is complete.

1.4.6 Add a new unit test module (for a new class under test)

To add unit tests for the methods in a new class follow the steps below:

1. Make a new file in the unit test folder named as mentioned in section *One unit test module per class under test*.
2. (Optional) Define an *init* function in the unit test module. Do this only if it is necessary to pass information to the tests at run time. Read more about the *init* function in the section *The init function*.
3. Add test case classes to this module. See *Add an unit test to an existing unit test module*.

Then it is necessary to make the unit test execution framework aware of your unit test module. Do this by making the following additions to the file `execute_unittests.py`.:

1. Import the class under test and the unit test module in the beginning of the file
2. Add an item to the `UNITTEST_MODULES` dict located right after the `### MAIN SCRIPT` comment. The added item should itself be a dictionary with items like this:

```
UNITTEST_MODULES = {
 'soco': {'name': 'SoCo', 'unittest_module': soco_unittest,
          'class': soco.SoCo, 'arguments': {'ip': ARGS.ip}},
 'foo_bar': {'name': 'FooBar', 'unittest_module': foo_bar_unittest,
            'class': soco.FooBar,'arguments': {'ip': ARGS.ip}}
 }
```

   where both the new imaginary `foo_bar` entry and the existing `soco` entry are shown for clarity. The arguments dict is what will be passed on to the `init` method, see section *The init function*.

3. Lastly, add the new module to the help text for the `modules` command line argument, defined in the `__build_option_parser` function:

```
parser.add_argument('--modules', type=str, default=None, help=''
                    'the modules to run unit test for can be '
                    '\'soco\', \'foo_bar\' or \'all\'')
```

   The name that should be added to the text is the key for the unit test module entry in the `UNITTEST_MODULES` dict.

The *init* function

Normally unit tests should be self-contained and therefore they should have all the data they will need built in. However, that does not apply to *SoCo*, because the IP's of the Sonos® units will be required and there is no way to know them in advance. Therefore, the execution script will call the function `init` in the unit test modules, if it exists, with a set of predefined arguments that can then be used for unit test initialization. Note that the function is to be named `init` , not `__init__` like the class initializers. The `init` function is called with one argument, which is the dictionary defined under the key `arguments` in the unit test modules definition. Please regard this as an exception to the general unit test best practices guidelines and use it only if there are no other option.

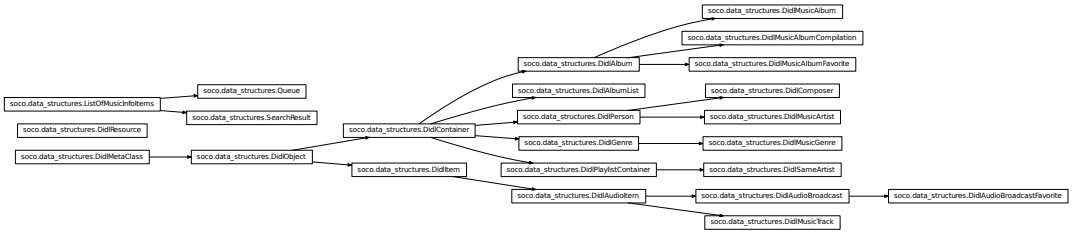
1.5 The `data_structures` sub module

1.5.1 Introduction

The majority of the data structures in this module are used to represent the metadata for music items, such as music tracks, genres and playlists. The data structure classes are documented in the sections below and the rest of this section contains a more thorough introduction.

Many music related items have a lot of metadata in common. For example, a music track and an album may both have artist and title metadata. It is possible therefore to derive a hierarchy of items, and to implement them as a class structure. The hierarchy which Sonos has adopted is represented by the DIDL Lite xml schema (DIDL stands for 'Digital Item Description Language'. For more details, see the UPnP specifications (PDF).

In the `data_structures` module, each class represents a particular DIDL-Lite object and is illustrated in *the figure below*. The black lines are the lines of inheritance, going from the top down.



All data structures are subclasses of the abstract *Didl Object item* class. You should never need to instantiate this directly. The subclasses are divided into *Containers* and *Items*. In general, *Containers* are things, like playlists, which are intended to contain other items.

At the bottom of the class hierarchy are 10 types of *DIDL items*. On each of these classes, relevant metadata items are available as attributes (though they may be implemented as properties). Each has a `title`, a `URI`, an `item id` and a *UPnP class*. Some have other attributes. For example, *DidlMusicTrack* and *DidlMusicAlbum* have some extra fields such as `album`, `album_art_uri` and `creator`.

One of the more important attributes which each class has is `didl_metadata`. It is used to produce the metadata that is sent to the Sonos® units in the form of XML. This metadata is created in an almost identical way for each class, which is why it is implemented in *DidlObject*. It uses the URI, the UPnP class and the title that the items are instantiated with, along with the two class variables `parent_id` and `_translation`.

1.5.2 Functions

`soco.data_structures.`**`ns_tag`**(*ns_id*, *tag*)

Return a namespace/tag item. The ns_id is translated to a full name space via the NAMESPACES variable.

1.5.3 DidlObject

class `soco.data_structures.`**`DidlObject`**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: `soco.data_structures.DidlMetaClass`

Abstract base class for all DIDL-Lite items.

You should not need to instantiate this.

`item_class`

str

The DIDL Lite class for this object

tag
str

The XML element tag name used for this instance

_translation
dict

A dict used to translate between instance attribute names and XML tags/namespaces. It also serves to define the allowed tags/attributes for this instance. Overridden and extended by subclasses.

__init__(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)
Construct and initialize a DidlObject.

Parameters

- **title** (*str*) – The title for the item
- **parent_id** (*str*) – The parent ID for the item
- **item_id** (*str*) – The ID for the item
- **restricted** (*bool*) – Whether the item can be modified
- **resources** (*list*) – A list of resources for this object
- **desc** (*str*) – A didl descriptor, default RINCON_AssociatedZPUDN. This is not the same as "description"! It is used for identifying the relevant music service
- ****kwargs** – Extra metadata. What is allowed depends on the _translation class attribute, which in turn depends on the DIDL class

__eq__(*playable_item*)
Compare with another `playable_item`.

Returns True if items are equal, else False

Return type (bool)

__repr__()
Return the repr value for the item.

The repr is of the form:

```
<class_name 'middle_part[0:40]' at id_in_hex>
```

where middle_part is either the title item in content, if it is set, or `str(content)`. The output is also cleared of non-ascii characters.

__str__()
Return the str value for the item:

```
<class_name 'middle_part[0:40]' at id_in_hex>
```

where middle_part is either the title item in content, if it is set, or `str(content)`. The output is also cleared of non-ascii characters.

__eq__(*playable_item*)
Compare with another `playable_item`.

Returns True if items are equal, else False

Return type (bool)

__init__(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)
Construct and initialize a DidlObject.

Parameters

- **title** (*str*) – The title for the item
- **parent_id** (*str*) – The parent ID for the item
- **item_id** (*str*) – The ID for the item
- **restricted** (*bool*) – Whether the item can be modified
- **resources** (*list*) – A list of resources for this object
- **desc** (*str*) – A didl descriptor, default RINCON_AssociatedZPUDN. This is not the same as "description"! It is used for identifying the relevant music service
- ****kwargs** – Extra metadata. What is allowed depends on the _translation class attribute, which in turn depends on the DIDL class

__ne__(*playable_item*)
Compare with another `playable_item`.

Returns True if items are unequal, else False

Return type (bool)

__repr__()
Return the repr value for the item.

The repr is of the form:

```
<class_name 'middle_part[0:40]' at id_in_hex>
```

where middle_part is either the title item in content, if it is set, or `str(content)`. The output is also cleared of non-ascii characters.

__str__()
Return the str value for the item:

```
<class_name 'middle_part[0:40]' at id_in_hex>
```

where middle_part is either the title item in content, if it is set, or `str(content)`. The output is also cleared of non-ascii characters.

classmethod from_dict(*content*)
Create an instance from a dict.

An alternative constructor. Equivalent to DidlObject(**content).

Arg: content (dict): Dict containing metadata information.Required and valid arguments are the same as for the `__init__` method.

classmethod from_element(*element*)
Create an instance of this class from an ElementTree xml Element.

An alternative constructor. The element must be a DIDL-Lite <item> or <container> element, and must be properly namespaced.

Arg: xml (Element): An `xml.etree.ElementTree.Element` object.

to_dict()
Return the dict representation of the instance.

to_element(*include_namespaces=False*)
Return an ElementTree Element representing this instance.

Arg:

include_namespaces (bool, optional): If True, include xml namespace attributes on the root element

Returns An ElementTree Element

```
<DIDL-Lite ..NS_INFO..>
  <item id="...self.item_id..."
    parentID="...cls.parent_id..." restricted="true">
    <dc:title>...self.title...</dc:title>
    <upnp:class>...self.item_class...</upnp:class>
    <desc id="cdudn"
      nameSpace="urn:schemas-rinconnetworks-com:metadata-1-0/">
      RINCON_AssociatedZPUDN
    </desc>
  </item>
</DIDL-Lite>
```

1.5.4 DidlContainer

class soco.data_structures.**DidlContainer**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)
Bases: *soco.data_structures.DidlObject*

Class that represents a music library container.

1.5.5 DidlItem

class soco.data_structures.**DidlItem**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)
Bases: *soco.data_structures.DidlObject*

A basic content directory item.

1.5.6 DidlMusicTrack

class soco.data_structures.**DidlMusicTrack**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)
Bases: soco.data_structures.DidlAudioItem

Class that represents a music library track.

1.5.7 DidlMusicAlbum

class soco.data_structures.**DidlMusicAlbum**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)
Bases: soco.data_structures.DidlAlbum

Class that represents a music library album.

1.5.8 DidlMusicArtist

class soco.data_structures.**DidlMusicArtist**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: soco.data_structures.DidlPerson

Class that represents a music library artist.

1.5.9 DidlMusicGenre

class soco.data_structures.**DidlMusicGenre**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: soco.data_structures.DidlGenre

Class that represents a music genre.

1.5.10 DidlAlbumList

class soco.data_structures.**DidlAlbumList**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: *soco.data_structures.DidlContainer*

Class that represents a music library album list.

1.5.11 DidlComposer

class soco.data_structures.**DidlComposer**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: soco.data_structures.DidlPerson

Class that represents a music library composer.

1.5.12 DidlPlaylistContainer

class soco.data_structures.**DidlPlaylistContainer**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: *soco.data_structures.DidlContainer*

Class that represents a music library play list.

1.5.13 DidlAudioBroadcast

class soco.data_structures.**DidlAudioBroadcast**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: soco.data_structures.DidlAudioItem

Class that represents an audio broadcast.

1.5.14 DidlContainer

class soco.data_structures.**DidlContainer**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc=u'RINCON_AssociatedZPUDN'*, ***kwargs*)

Bases: *soco.data_structures.DidlObject*

Class that represents a music library container.

1.6 SoCo releases

1.6.1 SoCo 0.11 release notes

SoCo 0.11 is a new version of the SoCo library. This release adds new features and fixes several bugs.

SoCo (Sonos Controller) is a simple Python class that allows you to programmatically control Sonos speakers.

New Features and Improvements

- The new properties `is_playing_tv`, `is_playing_radio` and `is_playing_line_in` have been added (#225)
- A method `get_item_album_art_uri` has been added to return the absolute album art full uri so that it is easy to put the album art in user interfaces (#240).
- Added support for satellite speaker detection in network topology parsing code (#245)
- Added support to search the music library for tracks, an artists' albums and an artist's album's tracks (#246)
- A fairly extensive re-organisation of the DIDL metadata handling code, which brings SoCo more into line with the DIDL-Lite spec, as adopted by Sonos. DIDL objects can have now have multiple URIs, and the interface is much simpler. (#256)
- Event objects now have a timestamp field (#273)
- The IP address (ie network interface) for discovering Sonos speakers can now be specified (#277)
- It is now possible to trigger an update of the music library (#286)
- The event listener port is now configurable (#288)
- Methods that can only be executed on master speakers will now raise a `SoCoSlaveException` (#296)
- An example has been added that shows how to play local files by setting up a temporary HTTP server in python (#307)
- Test cleanup (#309)

Bugfixes

- The value of the IP_MULTICAST_TTL option is now ensured to be one byte long (#269)
- Various encoding issues have been fixed (#293, #281, #306)
- Fix bug with browsing of imported playlists (#265)
- The `discover` method was broken in Python 3.4 (#271)
- An unknown / missing UPnP class in event subscriptions has been added (#266, #301, #303)
- Fix `add_to_queue` which was broken since the data structure refactoring (#308, #310)

Backwards Compatability

- The exception `DidlCannotCreateMetadata` has been deprecated. `DidlMetadataError` should be used instead. (#256)
- Code which has been deprecated for more than 3 releases has been removed. See previous release notes for deprecation notices. (#273)

1.6.2 SoCo 0.10 release notes

SoCo 0.10 is a new version of the SoCo library. This release adds new features and fixes several bugs.

SoCo (Sonos Controller) is a simple Python class that allows you to programmatically control Sonos speakers.

New Features

- Add support for taking a snapshot of the Sonos state, and then to restore it later (#224, #251)
- Added create_sonos_playlist_from_queue. Creates a new Sonos playlist from the current queue (#229)

Improvements

- Added a queue_size property to quickly return the size of the queue without reading any items (#217)
- Add metadata to return structure of get_current_track_info (#220)
- Add option to play_uri that allows for the item to be set and then optionally played (#219)
- Add option to play_uri that allows playing with a URI and title instead of metadata (#221)
- Get the item ID from the XML responses which enables adding tracks for music services such as Rhapsody which do not have all the detail in the item URI (#233)
- Added label and short_label properties, to provide a consistent readable label for group members (#228)
- Improved documentation (#248, #253, #259)
- Improved code examples (#250, #252)

Bugfixes

- Fixed a bug where `get_ml_item()` would fail if a radio station was played (#226)
- Fixed a timeout-related regression in `soco.discover()` (#244)
- Discovery code fixed to account for closing of multicast sockets by certain devices (#202, #201)
- Fixed a bug where sometimes zone groups would be created without a coordinator (#230)

Backwards Compatability

The metadata classes (`ML*`) have all been renamed (generally to `Didl*`), and aligned more closely with the underlying XML. The Music Services data structures (`MS*`) have been moved to their own module, and metadata for radio broadcasts is now returned properly (#243).

The `URI` class has been removed. As an alternative the method *`soco.SoCo.play_uri()`* can be used to enqueue and play an URI. The class `soco.data_structures.DIDLObject` can be used if an object is required.

Work is still ongoing on the metadata classes, so further changes should be expected.

1.6.3 SoCo 0.9 release notes

New Features

- Alarm configuration (#171)

```
>>> from soco.alarms import Alarm, get_alarms
>>> # create an alarm with default properties
>>> # my_device is the SoCo instance on which the alarm will be played
>>> alarm = Alarm(my_device)
>>> print alarm.volume
20
>>> print get_alarms()
set([])
>>> # save the alarm to the Sonos system
>>> alarm.save()
>>> print get_alarms()
set([<Alarm id:88@15:26:15 at 0x107abb090>])
>>> # update the alarm
>>> alarm.recurrence = "ONCE"
>>> # Save it again for the change to take effect
>>> alarm.save()
>>> # Remove it
>>> alarm.remove()
>>> print get_alarms()
set([])
```

- Methods for browsing the Music library (#192, #203, #208)

```
import soco
soc = soco.SoCo('...ipaddress..')
some_album = soc.get_albums()['item_list'][0]
tracks_in_that_album = soc.browse(some_album)
```

- Support for full Album Art URIs (#207)
- Support for music queues (#214)

```
queue = soco.get_queue()
for item in queue:
    print item.title

print queue.number_returned
print queue.total_matches
print queue.update_id
```

- Support for processing of LastChange events (#194)
- Support for write operations on Playlists (#198)

Improvements

- Improved test coverage (#159, #184)
- Fixes for Python 2.6 support (#175)
- Event-subscriptions can be auto-renewed (#179)
- The `SoCo` class can replaced by a custom implementation (#180)
- The cache can be globally disabled (#180)
- Music Library data structures are constructed for DIDL XML content (#191).
- Added previously removed support for PyPy (#205)
- All music library methods (`browse`, `get_tracks` etc. #203 and `get_queue` #214) now returns container objects instead of dicts or lists. The metadata is now available from these container objects as named attributes, so e.g. on a queue object you can access the size with `queue.total_matches`.

Backwards Compatability

- Music library methods return container objects instead of dicts and lists (see above). The old way of accessing that metadata (by dictionary type indexing), has been deprecated and is planned to be removed 3 releases after 0.9.

1.6.4 SoCo 0.8 release notes

New Features

- Re-added support for Python 2.6 (#154)
- Added `SoCo.get_sonos_playlists()` (#114)
- Added methods for working with speaker topology
- *`soco.SoCo.group`* retrieves the `soco.groups.ZoneGroup` to which the speaker belongs (#132). The group itself has a `soco.groups.ZoneGroup.member` attribute returning all of its members. Iterating directly over the group is possible as well.
- Speakers can be grouped using *`soco.SoCo.join()`* (#136):

```
z1 = SoCo('192.168.1.101')
z2 = SoCo('192.168.1.102')
z1.join(z2)
```

- *`soco.SoCo.all_zones`* and *`soco.SoCo.visible_zones`* return all and all visible zones, respectively.

- *soco.SoCo.is_bridge* indicates if the SoCo instance represents a bridge.
- *soco.SoCo.is_coordinator* indicates if the SoCo instance is a group coordinator (#166)
- A new soco.plugins.spotify.Spotify plugin allows querying and playing the Spotify music catalogue (#119):

```
from soco.plugins.spotify import Spotify
from soco.plugins.spotify import SpotifyTrack
# create a new plugin, pass the soco instance to it
myplugin = Spotify(device)
print 'index: ' + str(myplugin.add_track_to_queue(SpotifyTrack('
    spotify:track:20DfkHC5grnKNJCzZQB6KC')))
print 'index: ' + str(myplugin.add_album_to_queue(SpotifyAlbum('
    spotify:album:6a50SaJpvdWDp13t0wUcPU')))
```

- A soco.data_structures.URI item can be passed to add_to_queue which allows playing music from arbitrary URIs (#147)

```
import soco
from soco.data_structures import URI

soc = soco.SoCo('...ip_address...')
uri = URI('http://www.noiseaddicts.com/samples/17.mp3')
soc.add_to_queue(uri)
```

- A new include_invisible parameter to *soco.discover()* can be used to retrieve invisible speakers or bridges (#146)
- A new timeout parameter to *soco.discover()*. If no zones are found within timeout seconds None is returned. (#146)
- Network requests can be cached for better performance (#131).
- It is now possible to subscribe to events of a service using its *subscribe* method, which returns a *Subscription* object. To unsubscribe, call the *unsubscribe* method on the returned object. (#121, #130)
- Support for reading and setting crossfade (#165)

Improvements

- Performance improvements for speaker discovery (#146)
- Various improvements to the Wimp plugin (#140).
- Test coverage tracking using coveralls.io (#163)

Backwards Compatability

- Queue related use 0-based indexing consistently (#103)
- soco.SoCo.get_speakers_ip() is deprecated in favour of *soco.discover()* (#124)

1.6.5 SoCo 0.7 release notes

New Features

- All information about queue and music library items, like e.g. the title and album of a track, are now included in data structure classes instead of dictionaries (the classes are available in the *The data_structures sub module* sub-module). This advantages of this approach are:
  - The type of the item is identifiable by its class name
  - They have useful `__str__` representations and an `__equals__` method
  - Information is available as named attributes
  - They have the ability to produce their own UPnP meta-data (which is used by the `add_to_queue` method).

  See the Backwards Compatibility notice below.
- A webservice analyzer has been added in `dev_tools/analyse_ws.py` (#46).
- The commandline interface has been split into a separate project socos. It provides an command line interface on top of the SoCo library, and allows users to control their Sonos speakers from scripts and from an interactive shell.
- Python 3.2 and later is now supported in addition to 2.7.
- A simple version of the first plugin for the Wimp service has been added (#93).
- The new `soco.discover()` method provides an easier interface for discovering speakers in your network. `SonosDiscovery` has been deprecated in favour of it (see Backwards Compatability below).
- SoCo instances are now singletons per IP address. For any given IP address, there is only one SoCo instance.
- The code for generating the XML to be sent to Sonos devices has been completely rewritten, and it is now much easier to add new functionality. All services exposed by Sonos zones are now available if you need them (#48).

Backwards Compatability

> **Warning:** Please read the section below carefully when upgrading to SoCo 0.7.

Data Structures

The move to using **data structure classes** for music item information instead of dictionaries introduces some **backwards incompatible changes** in the library (see #83). The *get_queue* and *get_library_information* functions (and all methods derived from the latter) are affected. In the data structure classes, information like e.g. the title is now available as named attributes. This means that by the update to 0.7 it will also be necessary to update your code like e.g:

```
# Version < 0.7
for item in soco.get_queue():
    print item['title']
# Version >=0.7
for item in soco.get_queue():
    print item.title
```

SonosDiscovery

The `SonosDiscovery` class has been deprecated (see #80 and #75).

Instead of the following

```
>>> import soco
>>> d = soco.SonosDiscovery()
>>> ips = d.get_speaker_ips()
>>> for i in ips:
...         s = soco.SoCo(i)
...         print s.player_name
```

you should now write

```
>>> import soco
>>> for s in soco.discover():
...         print s.player_name
```

Properties

A number of methods have been replaced with properties, to simplify use (see #62)

For example, use

```
soco.volume = 30
soco.volume -=3
soco.status_light = True
```

instead of

```
soco.volume(30)
soco.volume(soco.volume()-3)
soco.status_light("On")
```

1.6.6 SoCo 0.6 release notes

New features

- **Music library information:** Several methods has been added to get information about the music library. It is now possible to get e.g. lists of tracks, albums and artists.
- **Raise exceptions on errors:** Several *SoCo* specific exceptions has been added. These exceptions are now raised e.g. when *SoCo* encounters communications errors instead of returning an error codes. This introduces a **backwards incompatible** change in *SoCo* that all users should be aware of.

For SoCo developers

- **Added plugin framework:** A plugin framework has been added to *SoCo*. The primary purpose of this framework is to provide a natural partition of the code, in which code that is specific to the individual music services is separated out into its own class as a plugin. Read more about the plugin framework in *the docs*.
- **Added unit testing framework:** A unit testing framework has been added to *SoCo* and unit tests has been written for 30% of the methods in the `SoCo` class. Please consider supplementing any new functionality with the appropriate unit tests and fell free to write unit tests for any of the methods that are still missing.

Coming next

- **Data structure change:** For the next version of SoCo it is planned to change the way SoCo handles data. It is planned to use classes for all the data structures, both internally and for in- and output. This will introduce a **backwards incompatible** change and therefore users of SoCo should be aware that extra work will be needed upon upgrading from version 0.6 to 0.7. The data structure changes will be described in more detail in the release notes for version 0.7.

1.7 Release Procedures

This document describes the necessary steps for creating a new release of SoCo.

1.7.1 Preparations

- Assign a version number to the release, according to semantic versioning. Tag names should be prefixed with `v`.
- Create a GitHub issue for the new version (eg Release 0.7 #108). This issue can be used to discuss included changes, the version number, etc.
- Create a milestone for the planned release (if it does not already exist). The milestone can be used to track issues relating to the release. All relevant issues should be assigned to the milestone.
- Create the release notes in `release_notes.html`.

1.7.2 Create and Publish

- Verify that all tests pass.
- Update the version number in `__init__.py` (see [example](https://github.com/SoCo/SoCo/commit/d35171213eabbc4)).
- Tag the current commit, eg

```
git tag -a v0.7 -m 'release version 0.7'
```

- Push the tag. This will create a new release on GitHub.

```
git push --tags
```

- Update the GitHub release using the release notes from the documentation. The release notes can be abbreviated if a link to the documentation is provided.
- Upload the release to PyPI.

```
python setup.py sdist bdist_wheel upload
```

- Enable doc builds for the newly released version on Read the Docs.

1.7.3 Wrap-Up

- Create the milestone for the next release (with the most likely version number) and close the milestone for the current release.
- Share the news!

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

B

C

D

F

G

I