SoCo (Sonos Controller) Documentation

Release 0.6

Rahim Sonawalla, et al.

June 05, 2014

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 `SonosDiscovery` class.

```
sd = SonosDiscovery()
ips = sd.get_speaker_ips()
```

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 Plugins

Plugins can extend the functionality of SoCo.

1.2.1 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.

1.2.2 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()
```

1.2.3 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

1.3 Unit tests

The unit tests written for the *SoCo* module implements elementary checks of whether the individual methods produce the expected results. 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.

1.3.1 Running the unit tests

To run the unit tests enter the `unittest` folder in the source code checkout and run the unit 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
```

1.3.2 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`

1.3.3 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.

1.3.4 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.

1.4 The soco module

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

class soco.**SonosDiscovery**
A simple class for discovering Sonos speakers.

Public functions: get_speaker_ips – Get a list of IPs of all zoneplayers.

class soco.**SoCo**(*speaker_ip*)
A simple class for controlling a Sonos speaker.

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. mute – Get or Set Mute (or unmute) the speaker. volume – Get or set the volume of the speaker. bass – Get or set the speaker's bass EQ. set_player_name – set the name of the Sonos Speaker treble – Set the speaker's treble EQ. set_play_mode – Change repeat and shuffle settings on the queue. set_loudness – Turn on (or off) the speaker's loudness compensation. switch_to_line_in – Switch the speaker's input to line-in. status_light – Turn on (or off) the Sonos status light. 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_folders – Get search folders from the music library 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 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. get_group_coordinator – Get the coordinator for a grouped collection of Sonos units. get_speakers_ip – Get the IP addresses of all the Sonos speakers in the network.

add_to_queue(*uri*)
Adds a given track to the queue.

Returns: If the Sonos speaker successfully added the track, returns the queue position of the track added.

Raises SoCoException (or a subclass) upon errors.

bass(*bass=None*)
Get or set the Sonos speaker's bass EQ.

Arguments: bass – A value between -10 and 10.

Returns: If the bass argument was specified: returns true if the Sonos speaker successfully set the bass EQ.

If the bass argument was not specified: returns the current base value.

Raises SoCoException (or a subclass) upon errors.

clear_queue()
Removes all tracks from the queue.

Returns: True if the Sonos speaker cleared the queue.

Raises SoCoException (or a subclass) upon errors.

get_album_artists(*start=0*, *max_items=100*)
Convinience method for: get_music_library_information('album_artists') Refer to the docstring for that method

get_albums(*start=0*, *max_items=100*)
Convinience method for: get_music_library_information('albums') Refer to the docstring for that method

get_artists(*start=0*, *max_items=100*)
Convinience method for: get_music_library_information('artists') Refer to the docstring for that method

get_composers(*start=0*, *max_items=100*)
Convinience method for: get_music_library_information('composers') 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(*start=0*, *max_items=100*)
Convinience method for: get_music_library_information('genres') Refer to the docstring for that method.

get_group_coordinator(*zone_name*, *refresh=False*)

Get the IP address of the Sonos system that is coordinator for the group containing zone_name

Code contributed by Aaron Daubman (daubman@gmail.com)

Arguments: zone_name – the name of the Zone to control for which you need the coordinator

refresh – Refresh the topology cache prior to looking for coordinator

Returns: The IP address of the coordinator or None of one can not be determined

get_music_library_information(*search_type*, *start=0*, *max_items=100*)
Retrieve information about the music library

Arguments: search The kind of information to retrieve. Can be one of:

'folders', 'artists', 'album_artists', 'albums', 'genres', 'composers', 'tracks' and 'playlists', where playlists are the imported file based playlists from the music library

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.

Returns a dictionary with metadata for the search, with the keys 'number_returned', 'update_id', 'total_matches' and an 'item' list with the search results. The search results are dicts that with the following exceptions all has the following keys 'title', 'res', 'class', 'parent_id', 'restricted', 'id', 'protocol_info'. The exceptions are; that the playlists item in the folder search has no res item; the album and track items has an extra 'creator' field and the track items has additional 'album', 'album_art_uri' and 'original_track_number' fields.

Raises SoCoException (or a subclass) upon errors.

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/ Probs to the authors of that project.

get_playlists(*start=0*, *max_items=100*)
Convinience method for: get_music_library_information('playlists') Refer to the docstring for that method

get_queue(*start=0*, *max_items=100*)
Get information about the queue.

Returns: A list containing a dictionary for each track in the queue. The track dictionary contains the following information about the track: title, artist, album, album_art, uri

If we're unable to return data for a field, we'll return an empty list. This can happen for all kinds of reasons so be sure to check values.

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

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_speakers_ip(*refresh=False*)
Get the IP addresses of all the Sonos speakers in the network.

Code contributed by Thomas Bartvig (thomas.bartvig@gmail.com)

Arguments: refresh – Refresh the speakers IP cache.

Returns: IP addresses of the Sonos speakers.

get_tracks(*start=0*, *max_items=100*)
Convinience method for: get_music_library_information('tracks') Refer to the docstring for that method

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

Code contributed by Thomas Bartvig (thomas.bartvig@gmail.com)

Returns: True if this speaker has joined the master speaker

Raises SoCoException (or a subclass) upon errors.

mute(*mute=None*)
Mute or unmute the Sonos speaker.

Arguments: mute – True to mute. False to unmute.

Returns: True if the Sonos speaker was successfully muted or unmuted.

If the mute argument was not specified: returns the current mute status 0 for unmuted, 1 for muted

Raises SoCoException (or a subclass) upon errors.

next()
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://travelmarx.blogspot.dk/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.

Code contributed by Thomas Bartvig (thomas.bartvig@gmail.com)

Returns: True if partymode is set

Raises SoCoException (or a subclass) upon errors.

pause()
Pause the currently playing track.

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

Raises SoCoException (or a subclass) upon errors.

play()
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(*queue_index*)
Play an item from the queue. The track number is required as an argument, where the first track is 0.

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

Raises SoCoException (or a subclass) upon errors.

play_uri(*uri=''*, *meta=''*)
Play a given stream. Pauses the queue.

Arguments: uri – URI of a stream to be played. meta — The track metadata to show in the player, DIDL format.

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

Raises SoCoException (or a subclass) upon errors.

previous()
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.

remove_from_queue(*index*)
Removes a track from the queue.

index: the index of the track to remove; first item in the queue is 1

Returns: True if the Sonos speaker successfully removed the track

Raises SoCoException (or a subclass) upon errors.

seek(*timestamp*)
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.

set_loudness(*loudness*)
Set the Sonos speaker's loudness compensation.

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

Arguments: loudness – True to turn on loudness compensation. False to disable it.

Returns: True if the Sonos speaker successfully set the loundess compensation.

Raises SoCoException (or a subclass) upon errors.

set_play_mode(*playmode*)
Sets the play mode for the queue. 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.

Returns: True if the play mode was successfully set.

Raises SoCoException (or a subclass) upon errors.

set_player_name(*playername*)
Sets the name of the player

Returns: True if the player name was successfully set.

Raises SoCoException (or a subclass) upon errors.

status_light(*led_on*)
Turn on (or off) the white Sonos status light.

Turns on or off the little white light on the Sonos speaker. (It's between the mute button and the volume up button on the speaker.)

Arguments: led_on – True to turn on the light. False to turn off the light.

Returns: True if the Sonos speaker successfully turned on (or off) the light.

Raises SoCoException (or a subclass) upon errors.

stop()
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.

treble(*treble=None*)
Get or set the Sonos speaker's treble EQ.

Arguments: treble – A value between -10 and 10.

Returns: If the treble argument was specified: returns true if the Sonos speaker successfully set the treble EQ.

If the treble argument was not specified: returns the current treble value.

Raises SoCoException (or a subclass) upon errors.

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.

volume(*volume=None*)
Get or set the Sonos speaker volume.

Arguments: volume – A value between 0 and 100.

Returns: If the volume argument was specified: returns true if the Sonos speaker successfully set the volume.

If the volume argument was not specified: returns the current volume of the Sonos speaker.

Raises SoCoException (or a subclass) upon errors.

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

1.5 Release notes

1.5.1 Version 0.6

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.

CHAPTER 2

Indices and tables

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

S