SoCo (Sonos Controller) Documentation

Release 0.8

Rahim Sonawalla, et al.

September 12, 2014

Contents

1 Contents **3**
- 1.1 Tutorial . . . 3
- 1.2 The `soco` module . . . 3
- 1.3 Plugins . . . 10
- 1.4 Unit and integration tests . . . 11
- 1.5 The `data_structures` sub module . . . 14
- 1.6 Release notes . . . 23
- 1.7 Release Procedures . . . 27

2 Indices and tables **29**

Python Module Index **31**

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 The `soco` module

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

`soco.`**`discover`**(*timeout=1*, *include_invisible=False*)
Discover Sonos zones on the local network.

Return an set of visible 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.

class `soco.`**`SonosDiscovery`**
Retained for backward compatibility only. Will be removed in future releases

Deprecated since version 0.7: Use `discover()` instead.

static `get_speaker_ips`()
Deprecated in favour of discover()

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 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_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_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.
```

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.
status_light -- The state of the Sonos status light.
player_name  -- The speaker's name.
play_mode -- The queue's repeat/shuffle settings.
```

> **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_to_queue(*queueable_item*)
Adds a queueable item to the queue

add_uri_to_queue(*uri*)
Adds the URI to the queue

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

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.

clear_queue()
Removes all tracks from the queue.

Returns: True if the Sonos speaker cleared the queue.

Raises SoCoException (or a subclass) upon errors.

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

get_album_artists(*start=0*, *max_items=100*)
Convinience 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(*start=0*, *max_items=100*)
Convinience method for `get_music_library_information()` with *search_type='albums'*. For details on remaining arguments refer to the docstring for that method.

get_artists(*start=0*, *max_items=100*)
Convinience method for `get_music_library_information()` with *search_type='artists'*. For details on remaining arguments refer to the docstring for that method.

get_composers(*start=0*, *max_items=100*)
Convinience 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(*start=0*, *max_items=100*)
Convinience method for `get_music_library_information()` with *search_type='genres'*. For details on remaining arguments refer to the docstring for that method.

get_group_coordinator(*zone_name*)
Deprecated since version 0.8.

Use `group()` or `all_groups()` instead.

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

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
- **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' list with the search results. The search results are instances of one of `MLArtist`, `MLAlbumArtist`, `MLAlbum`, `MLGenre`, `MLComposer`, `MLTrack`, `MLShare`, `MLPlaylist` depending on the type of the search.

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

Parameters

- **start** – Starting number of returned matches
- **max_items** – Maximum number of returned matches

Returns A list of `QueueItem`.

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

get_sonos_playlists(*start=0*, *max_items=100*)
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_speakers_ip(*refresh=False*)
Get the IP addresses of all the Sonos speakers in the network.

Arguments: refresh – Refresh the speakers IP cache. Ignored. For backward

compatibility only

Returns: a set of IP addresses of the Sonos speakers.

Deprecated since version 0.8.

get_tracks(*start=0*, *max_items=100*)
Convinience method for `get_music_library_information()` with *search_type='tracks'*. For details on remaining arguments refer to the docstring for that method.

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_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

---

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()
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()
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(*index*)
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

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

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(*uri=u''*, *meta=u''*)
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.

player_name
The speaker's name. A string.

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*)
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.

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.

speaker_ip
Retained for backward compatibility only. Will be removed in future releases

Deprecated since version 0.7: Use `ip_address` instead.

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()
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 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

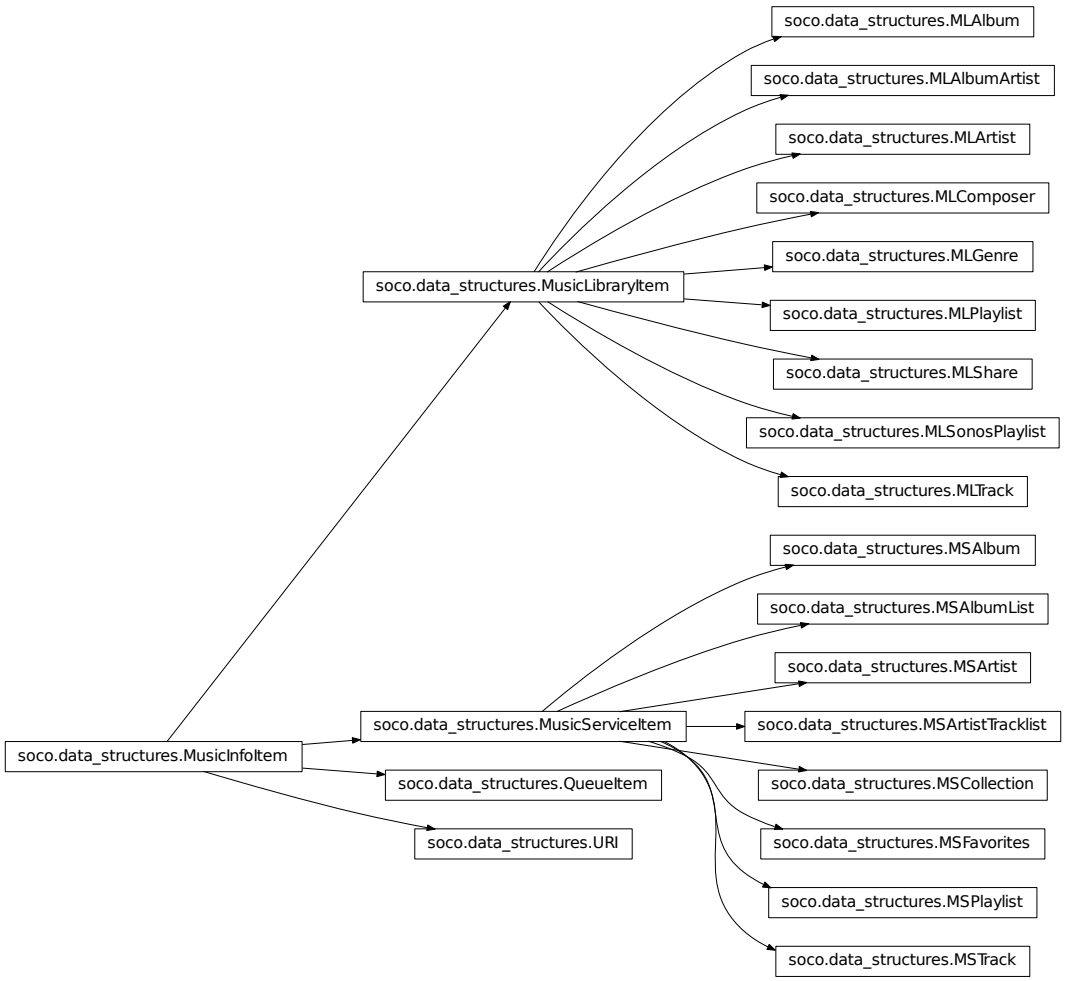
The data structures are used to represent playable items like e.g. a music track or playlist. The data structure classes are documented in the sections below and the rest of this section contains a more thorough introduction.

To expand a bit, the `data_structures` sub-module consist of a hierarchy of classes that represent different music information items. This could be a "real" item such as a music library track, album or genre or an abstract item such as a music library item.

The main advantages of using classes as apposed to e.g. dicts to contain the information are:

- They are easy to identify
- It is possibly to define and agree on certain abilities such as what is the criteria for two tracks being equal
- Certain functionality for these information object, such as producing the XML that is needed for the UPnP communication can be attached to the elements themselves.

Many of the items have a lot in common and therefore has shared functionality. This has been implemented by means of inheritance, in such a way that common functionality is always pulled up the inheritance hierarchy to the highest point that have this functionality in common. The hierarchy is illustrated in figure *the figure below*. The black lines are the lines of inheritance, going from the top down.



All data structures are `music information items`. Three classes inherit from this top level class; the `queue item`, the `music library item` and the music service item

There are 8 types of `music library items`, represented by the 8 classes that inherit from it. From these classes all information items are available as named properties. All of these items contains a `title`, a `URI` and a `UPnP class`, so these items are defined in the `MusicLibraryItem` class and inherited by them all. For most items the `ID` can be extracted from the URI in the same way, so therefore it is defined in `MusicLibraryItem.item_id` and the few classes (`MLTrack`, `MLPlaylist`) that extract the ID differently from the URI then overrides this property. Besides the information items that they all share, `MLTrack` and `MLAlbum` define some extra fields such as `album`, `album_art_uri` and `creator`.

One of the more important attributes is `didl_metadata`. It is used to produce the metadata that is sent to the Sonos® units. This metadata is created in an almost identical way, which is the reason that it is implemented in `MusicLibraryItem`. It uses the URI (through the ID), the UPnP class and the title that the items are instantiated with and the two class variables `parent_id` and `_translation`. `parent_id` must be over written in each of the sub classes, whereas that is only necessary for `_translation` if the information fields are different from the default.

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 NS module variable.

soco.data_structures.**get_ml_item**(*xml*)
Return the music library item that corresponds to xml. The class is identified by getting the parentID and making a lookup in the PARENT_ID_TO_CLASS module variable dictionary.

1.5.3 MusicInfoItem

class soco.data_structures.**MusicInfoItem**
Bases: object

Abstract class for all data structure classes

__init__()
Initialize the content as an empty dict.

__eq__(*playable_item*)
Return the equals comparison result to another playable_item.

__repr__()
Return the repr value for the item.

The repr is on 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.

1.5.4 MusicLibraryItem

class soco.data_structures.**MusicLibraryItem**(*uri*, *title*, *item_class*, ***kwargs*)
Bases: soco.data_structures.MusicInfoItem

Abstract class for a queueable item from the music library.

Variables

- **parent_id** – The parent ID for the music library item is None, since it is a abstract class and it should be overwritten in the sub classes
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MusicLibraryItems from XML. The default value is shown below. This default value applies to most sub classes and the rest should overwrite it.

```
    # key: (ns, tag)
    _translation = {
        'title': ('dc', 'title'),
        'uri': ('', 'res'),
        'item_class': ('upnp', 'class')
    }
```

__init__(*uri*, *title*, *item_class*, ***kwargs*)
Initialize the MusicLibraryItem from parameter arguments.

Parameters

- **uri** – The URI for the item
- **title** – The title for the item
- **item_class** – The UPnP class for the item
- ****kwargs** – Extra information items to form the music library item from. Valid keys are `album`, `album_art_uri`, `creator` and `original_track_number`. `original_track_number` is an int, all other values are unicode objects.

didl_metadata
Produce the DIDL metadata XML.

This method uses the `item_id` attribute (and via that the `uri` attribute), the `item_class` attribute and the `title` attribute. The metadata will be on the form:

```
<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>
```

classmethod from_dict(*content*)
Return an instance of this class, created from a dict with parameters.

Parameters **content** – Dict with information for the music library item. Required and valid arguments are the same as for the `__init__` method.

classmethod from_xml(*xml*)
Return an instance of this class, created from xml.

Parameters **xml** – An `xml.etree.ElementTree.Element` object. The top element usually is a DIDL-LITE item (NS['']) element. Inside the item element should be the (namespace, tag_name) elements in the dictionary-key-to-xml-tag-and-namespace-translation described in the class docstring.

item_class
Get and set the UPnP object class as an unicode object.

item_id
Return the id.

The id is extracted as the part of the URI after the first # character. For the few music library types where that is not correct, this method should be overwritten.

title
Get and set the title as an unicode object.

to_dict
Get the dict representation of the instance.

uri
Get and set the URI as an unicode object.

1.5.5 MLTrack

class soco.data_structures.**MLTrack**(*uri*, *title*, *item_class=u'object.item.audioItem.musicTrack'*, ***kwargs*)
Bases: soco.data_structures.MusicLibraryItem

Class that represents a music library track.

Variables

- **parent_id** – The parent ID for the MLTrack is 'A:TRACKS'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLTrack from XML. The value is shown below

```
# key: (ns, tag)
_translation = {
    'title': ('dc', 'title'),
    'creator': ('dc', 'creator'),
    'album': ('upnp', 'album'),
    'album_art_uri': ('upnp', 'albumArtURI'),
    'uri': ('', 'res'),
    'original_track_number': ('upnp', 'originalTrackNumber'),
    'item_class': ('upnp', 'class')
}
```

__init__(*uri*, *title*, *item_class=u'object.item.audioItem.musicTrack'*, ***kwargs*)
Instantiate the MLTrack item by passing the arguments to the super class MusicLibraryItem.__init__().

Parameters

- **uri** – The URI for the track
- **title** – The title of the track
- **item_class** – The UPnP class for the track. The default value is: `object.item.audioItem.musicTrack`
- ****kwargs** – Optional extra information items, valid keys are: `album`, `album_art_uri`, `creator`, `original_track_number`. `original_track_number` is an `int`. All other values are unicode objects.

album
Get and set the album as an unicode object.

album_art_uri
Get and set the album art URI as an unicode object.

creator
Get and set the creator as an unicode object.

item_id
Return the id.

original_track_number
Get and set the original track number as an `int`.

1.5.6 MLAlbum

class soco.data_structures.**MLAlbum**(*uri*, *title*, *item_class=u'object.container.album.musicAlbum'*, ***kwargs*)
Bases: soco.data_structures.MusicLibraryItem

Class that represents a music library album.

Variables

- **parent_id** – The parent ID for the MLTrack is 'A:ALBUM'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLAlbum from XML. The value is shown below

```
# key: (ns, tag)
_translation = {
    'title': ('dc', 'title'),
    'creator': ('dc', 'creator'),
    'album_art_uri': ('upnp', 'albumArtURI'),
    'uri': ('', 'res'),
    'item_class': ('upnp', 'class')
}
```

__init__(*uri*, *title*, *item_class=u'object.container.album.musicAlbum'*, ***kwargs*)
Instantiate the MLAlbum item by passing the arguments to the super class MusicLibraryItem.__init__().

Parameters

- **uri** – The URI for the alum
- **title** – The title of the album
- **item_class** – The UPnP class for the album. The default value is: `object.container.album.musicAlbum`
- ****kwargs** – Optional extra information items, valid keys are: `album_art_uri` and `creator`. All value should be unicode objects.

album_art_uri
Get and set the album art URI as an unicode object.

creator
Get and set the creator as an unicode object.

1.5.7 MLArtist

class soco.data_structures.**MLArtist**(*uri*, *title*, *item_class=u'object.container.person.musicArtist'*)
Bases: soco.data_structures.MusicLibraryItem

Class that represents a music library artist.

Variables

- **parent_id** – The parent ID for the MLArtist is 'A:ARTIST'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLArtist from XML is inherited from `MusicLibraryItem`.

__init__(*uri*, *title*, *item_class=u'object.container.person.musicArtist'*)
Instantiate the MLArtist item by passing the arguments to the super class `MusicLibraryItem.__init__()`.

Parameters

- **uri** – The URI for the artist
- **title** – The title of the artist
- **item_class** – The UPnP class for the artist. The default value is: `object.container.person.musicArtist`

1.5.8 MLAlbumArtist

class `soco.data_structures.`**MLAlbumArtist**(*uri*, *title*, *item_class=u'object.container.person.musicArtist'*)
Bases: `soco.data_structures.MusicLibraryItem`

Class that represents a music library album artist.

Variables

- **parent_id** – The parent ID for the MLAlbumArtist is 'A:ALBUMARTIST'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLAlbumArtist from XML is inherited from `MusicLibraryItem`.

__init__(*uri*, *title*, *item_class=u'object.container.person.musicArtist'*)
Instantiate the MLAlbumArtist item by passing the arguments to the super class `MusicLibraryItem.__init__()`.

Parameters

- **uri** – The URI for the alum artist
- **title** – The title of the album artist
- **item_class** – The UPnP class for the album artist. The default value is: `object.container.person.musicArtist`

1.5.9 MLGenre

class `soco.data_structures.`**MLGenre**(*uri*, *title*, *item_class=u'object.container.genre.musicGenre'*)
Bases: `soco.data_structures.MusicLibraryItem`

Class that represents a music library genre.

Variables

- **parent_id** – The parent ID for the MLGenre is 'A:GENRE'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLGenre from XML is inherited from `MusicLibraryItem`.

__init__(*uri*, *title*, *item_class=u'object.container.genre.musicGenre'*)
Instantiate the MLGenre item by passing the arguments to the super class `MusicLibraryItem.__init__()`.

Parameters

- **uri** – The URI for the genre
- **title** – The title of the genre
- **item_class** – The UPnP class for the genre. The default value is: `object.container.genre.musicGenre`

1.5.10 MLComposer

class `soco.data_structures.`**`MLComposer`**(*uri*, *title*, *item_class=u'object.container.person.composer'*)
Bases: `soco.data_structures.MusicLibraryItem`

Class that represents a music library composer.

Variables

- **parent_id** – The parent ID for the MLComposer is 'A:COMPOSER'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLComposer from XML is inherited from `MusicLibraryItem`.

`__init__`(*uri*, *title*, *item_class=u'object.container.person.composer'*)
Instantiate the MLComposer item by passing the arguments to the super class `MusicLibraryItem.__init__()`.

Parameters

- **uri** – The URI for the composer
- **title** – The title of the composer
- **item_class** – The UPnP class for the composer. The default value is: `object.container.person.composer`

1.5.11 MLPlaylist

class `soco.data_structures.`**`MLPlaylist`**(*uri*, *title*, *item_class=u'object.container.playlistContainer'*)
Bases: `soco.data_structures.MusicLibraryItem`

Class that represents a music library play list.

Variables

- **parent_id** – The parent ID for the MLPlaylist is 'A:PLAYLIST'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLPlaylist from XML is inherited from `MusicLibraryItem`.

`__init__`(*uri*, *title*, *item_class=u'object.container.playlistContainer'*)
Instantiate the MLPlaylist item by passing the arguments to the super class `MusicLibraryItem.__init__()`.

Parameters

- **uri** – The URI for the playlist
- **title** – The title of the playlist
- **item_class** – The UPnP class for the playlist. The default value is: `object.container.playlistContainer`

item_id
Returns the id.

1.5.12 MLShare

class soco.data_structures.**MLShare**(*uri*, *title*, *item_class=u'object.container'*)
Bases: soco.data_structures.MusicLibraryItem

Class that represents a music library share.

Variables

- **parent_id** – The parent ID for the MLShare is 'S:'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a MLShare from XML is inherited from MusicLibraryItem.

__init__(*uri*, *title*, *item_class=u'object.container'*)
Instantiate the MLShare item by passing the arguments to the super class MusicLibraryItem.__init__().

Parameters

- **uri** – The URI for the share
- **title** – The title of the share
- **item_class** – The UPnP class for the share. The default value is: object.container

1.5.13 QueueItem

class soco.data_structures.**QueueItem**(*uri*, *title*, *item_class=u'object.item.audioItem.musicTrack'*, ***kwargs*)
Bases: soco.data_structures.MusicInfoItem

Class that represents a queue item.

Variables

- **parent_id** – The parent ID for the QueueItem is 'Q:0'
- **_translation** – The dictionary-key-to-xml-tag-and-namespace- translation used when instantiating a QueueItem from XML. The value is shown below

```
# key: (ns, tag)
_translation = {
    'title': ('dc', 'title'),
    'creator': ('dc', 'creator'),
    'album': ('upnp', 'album'),
    'album_art_uri': ('upnp', 'albumArtURI'),
    'uri': ('', 'res'),
    'original_track_number': ('upnp', 'originalTrackNumber'),
    'item_class': ('upnp', 'class')
}
```

__init__(*uri*, *title*, *item_class=u'object.item.audioItem.musicTrack'*, ***kwargs*)
Instantiate the QueueItem by passing the arguments to the super class MusicInfoItem.__init__().

Parameters

- **uri** – The URI for the queue item

- **title** – The title of the queue item
- **item_class** – The UPnP class for the queue item. The default value is: `object.item.audioItem.musicTrack`
- ****kwargs** – Optional extra information items, valid keys are: `album`, `album_art_uri`, `creator`, `original_track_number`. `original_track_number` is an `int`. All other values are unicode objects.

album
Get and set the album as an unicode object.

album_art_uri
Get and set the album art URI as an unicode object.

creator
Get and set the creator as an unicode object.

didl_metadata
Produce the DIDL metadata XML.

classmethod from_dict(*content*)
Return an instance of this class, created from a dict with parameters.

Parameters **content** – Dict with information for the music library item. Required and valid arguments are the same as for the `__init__` method.

classmethod from_xml(*xml*)
Return an instance of this class, created from xml.

Parameters **xml** – An `xml.etree.ElementTree.Element` object. The top element usually is a DIDL-LITE item (NS['']) element. Inside the item element should be the (namespace, tag_name) elements in the dictionary-key-to-xml-tag-and-namespace-translation described in the class docstring.

item_class
Get and set the UPnP object class as an unicode object.

original_track_number
Get and set the original track number as an `int`.

title
Get and set the title as an unicode object.

to_dict
Get the dict representation of the instance.

uri
Get and set the URI as an unicode object.

1.6 Release notes

1.6.1 Version 0.8

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)

1.6.2 Version 0.7

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")
```

1.6.3 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.

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

soco, 3