soco Documentation

Release 0.12

Author

October 09, 2016

SoCo (Sonos Controller) is a high level Python 2/3 library to control your Sonos ® speakers with:

```
# Import soco and get a SoCo instance
import soco
device = soco.discovery.any_soco()

# Get all albums from the music library that contains the word "Black"
# and add them to the queue
albums = device.music_library.get_albums(search_term='Black')
for album in albums:
    print('Added:', album.title)
    device.add_to_queue(album)

# Dial up the volume (just a bit) and play
device.volume += 10
device.play()
```

To get up and running quickly with *SoCo*, start by reading the *getting started* page, with *installation instructions* and a small *tutorial* and then wet your appetite with the *micro examples*. Then optionally follow up with any of the advanced topics that peak your interest: *Speaker Topologies*, *Events* and *UPnP Services*. Finally dive into the *the full module reference documentation*.

For support post a question in the SoCo Google group or file an issue on Github.

If you are interested in participating in the development, plase read *the development documentation* and file a bug or make a pull request on Github.

CHAPTER 1

Contents

1.1 Getting started

This section will help you to quickly get started with *SoCo*.

1.1.1 Installation

SoCo can be installed either with *pip* (recommended) or *manually*.

From PyPi with pip

The easiest way to install *SoCo*, is to install it from PyPi with the program pip. This can be done with the command:

```
pip install soco
```

This will automatically take care of installing any dependencies you need.

Manual installation from .tar.gz file

SoCo can also be installed manually from the .tar.gz file. First, find the latest version of SoCo on PyPi and download the `.tar.gz` file at the bottom of the page. After that, extract the content and move into the extracted folder. As an example, for *SoCo* 0.11.1 and on a Unix type system, this can be done with the following commands:

```
wget https://pypi.python.org/packages/source/s/soco/soco-0.11.1.tar.gz#md5=73187104385f04d18ce3e5685
tar zxvf soco-0.11.1.tar.gz
cd soco-0.11.1/
```

Have a look inside the `requirements.txt` file. You will need to install the dependencies listed in that file yourself. See the documentation for the individual dependencies for installation instructions.

After the requirements are in place, the package can be install with the command:

```
python setup.py install
```

After installation check

After installation, open a Python interpreter and check that `soco` can be imported and that your Sonos® players can be discovered:

```
>>> import soco
>>> soco.discover()
set([SoCo("192.168.0.16"), SoCo("192.168.0.17"), SoCo("192.168.0.10")])
```

1.1.2 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.

Discovery

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

```
>>> import soco
>>> speakers = soco.discover()
```

It returns a `set` of `soco.SoCo` instances, each representing a speaker in your network.

Music

You can use those SoCo instances to inspect and interact with your speakers.

```
>>> speaker = speakers.pop()
>>> speaker.player_name
'Living Room'
>>> speaker.ip_address
u'192.168.0.129'

>>> speaker.volume
10
>>> speaker.volume = 15
>>> speaker.play()
```

See for `soco.SoCo` for all methods that are available for a speaker.

1.2 Examples

This page contains collection of small examples to show of the features of *SoCo* and hopefully get you well started with the library.

All examples are shown as if entered in the Python interpreter (as apposed to executed from a file) because that makes it easy to incorporate output in the code listings.

All the examples from *Playback control* and forward assume that you have followed one of the examples in *Getting your devices* and therefore already have a variable named `device` that points to a `soco.SoCo` instance.

1.2.1 Getting your devices

Getting all your devices

To get all your devices use the `soco.discover()` function:

```
>>> import soco
>>> devices = soco.discover()
>>> devices
set([SoCo("192.168.0.10"), SoCo("192.168.0.30"), SoCo("192.168.0.17")])
>>> device = devices.pop()
>>> device
SoCo("192.168.0.16")
```

Getting any device

To get any device use the *soco.discovery.any_soco()* function. This can be useful for cases where you really do not care which one you get, you just need one e.g. to query for music library information:

```
>>> import soco
>>> device = soco.discovery.any_soco()
>>> device
SoCo("192.168.0.16")
```

Getting a named device

Getting a device by name is done by searching through the devices returned by `soco.discover()`:

```
>>> import soco
>>> for device in soco.discover():
...     if device.player_name == 'Office':
...         break
... else:
...     device = None
...
>>> device
SoCo("192.168.1.8")
```

1.2.2 Playback control

Play, pause and stop

The normal play, pause and stop functionality is provided with similarly named methods (*play()*, *pause()* and *stop()*) on the *SoCo* instance and the current state is included in the output of *get_current_transport_info()*:

```
>>> device.get_current_transport_info()['current_transport_state']
'STOPPED'
>>> device.play()
>>> device.get_current_transport_info()['current_transport_state']
'PLAYING'
>>> device.pause()
>>> device.get_current_transport_info()['current_transport_state']
'PAUSED_PLAYBACK'
```

More playback control with next, previous and seek

Navigating to the next or previous track is similarly done with methods of the same name (*next()* and *previous()*) and information about the current position in the queue is contained in the output from

get_current_track_info():

```
>>> device.get_current_track_info()['playlist_position']
'29'
>>> device.next()
>>> device.get_current_track_info()['playlist_position']
'30'
>>> device.previous()
>>> device.get_current_track_info()['playlist_position']
'29'
```

Seeking is done with the *seek()* method. Note that the input for that method is a string on the form "HH:MM:SS" or "H:MM:SS". The current position is also contained in *get_current_track_info()*:

```
>>> device.get_current_track_info()['position']
'0:02:59'
>>> device.seek("0:00:30")
>>> device.get_current_track_info()['position']
'0:00:31'
```

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.

`name`
human-readable name of the plugin

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

1.4 Authors

1.4.1 Project Creator

SoCo was created in 2012 at Music Hack Day Sydney by Rahim Sonawalla

1.4.2 Maintainers

- Lawrence Akka
- Stefan Kögl
- Kenneth Nielsen

1.4.3 Contributors

(alphabetical)

- Petter Aas
- Murali Allada
- Joel Björkman
- Aaron Daubman
- Johan Elmerfjord
- David H
- Jeff Hinrichs
- Jeroen Idserda
- Todd Neal
- nixscripter
- Kenneth Nielsen
- Dave O'Connor
- Dennnis O'Reilly
- phut
- Dan Poirier
- Jason Ting

- Scott G Waters

1.5 Speaker Topologies

Sonos speakers can be grouped together, and existing groups can be inspected.

Topology is available from each `soco.SoCo` instance.

```
>>> my_player.group
ZoneGroup(
    uid='RINCON_000E5879136C01400:58',
    coordinator=SoCo("192.168.1.101"),
    members=set([SoCo("192.168.1.101"), SoCo("192.168.1.102")])
)
```

A group of speakers is represented by a *soco.groups.ZoneGroup*.

1.5.1 Zone Group

Each `ZoneGroup` contains its coordinator

```
>>> my_player.group.coordinator
SoCo("192.168.1.101")
```

which is again a `soco.SoCo` instance

```
>>> my_player.group.coordinator.player_name
Kitchen
```

A `ZoneGroup` also contains a set of members.

```
>>> my_player.group.members
{SoCo("192.168.1.101"), SoCo("192.168.1.102")}
```

For convenience, `ZoneGroup` is also a container:

```
>>> for player in my_player.group:
...   print(player.player_name)
Living Room
Kitchen
```

If you need it, you can get an iterator over all groups on the network:

```
>>> my_player.all_groups
<generator object all_groups at 0x108cf0c30>
```

1.6 UPnP Services

Sonos devices offer several UPnP services which are accessible from classes in the *soco.services* module.

- *soco.services.AlarmClock*
- *soco.services.MusicServices*
- *soco.services.DeviceProperties*
- *soco.services.SystemProperties*

- *soco.services.ZoneGroupTopology*
- *soco.services.GroupManagement*
- *soco.services.QPlay*
- *soco.services.ContentDirectory*
- *soco.services.MS_ConnectionManager*
- *soco.services.RenderingControl*
- *soco.services.MR_ConnectionManager*
- *soco.services.AVTransport*
- *soco.services.Queue*
- *soco.services.GroupRenderingControl*

All services take a `soco.SoCo` instance as their first parameter.

1.6.1 Inspecting

To get a list of supported actions you can call the service's *soco.services.Service.iter_actions()*. It yields the service's actions with their in_arguments (ie parameters to pass to the action) and out_arguments (ie returned values).

Each action is an *soco.services.Action* namedtuple, consisting of `action_name` (a string), `in_args` (a list of *soco.services.Argument* namedtuples consisting of `name` and `argtype`), and out_args (ditto), eg:

1.6.2 Events

You can subscribe to the events of a service using the *soco.services.Service.subscribe()* method. See *Events* for details.

1.7 Events

You can receive events about changes on the Sonos network.

The *soco.services.Service.subscribe()* method of a service now returns a *soco.events.Subscription* object. To unsubscribe, call the *soco.events.Subscription.unsubscribe()* method on the returned object.

Each subscription has its own queue. Events relevant to that subscription are put onto that queue, which can be accessed from `subscription.events.get()`.

Some XML parsing is done for you when you retrieve an event from the event queue. The `get` and `get_nowait` methods will return a dict with keys which are the evented variables and values which are the values sent by the event.

1.7.1 Example

```
try:
    from queue import Empty
except:  # Py2.7
    from Queue import Empty
```

```
import soco
from soco.events import event_listener
import logging
logging.basicConfig(level=logging.DEBUG)
# pick a device
device = soco.discover().pop()
# Subscribe to ZGT events
sub = device.zoneGroupTopology.subscribe()

# print out the events as they arise
while True:
    try:
        event = sub.events.get(timeout=0.5)
        print(event)
        print(event.sid)
        print(event.seq)

    except Empty:
        pass
    except KeyboardInterrupt:
        sub.unsubscribe()
        event_listener.stop()
        break
```

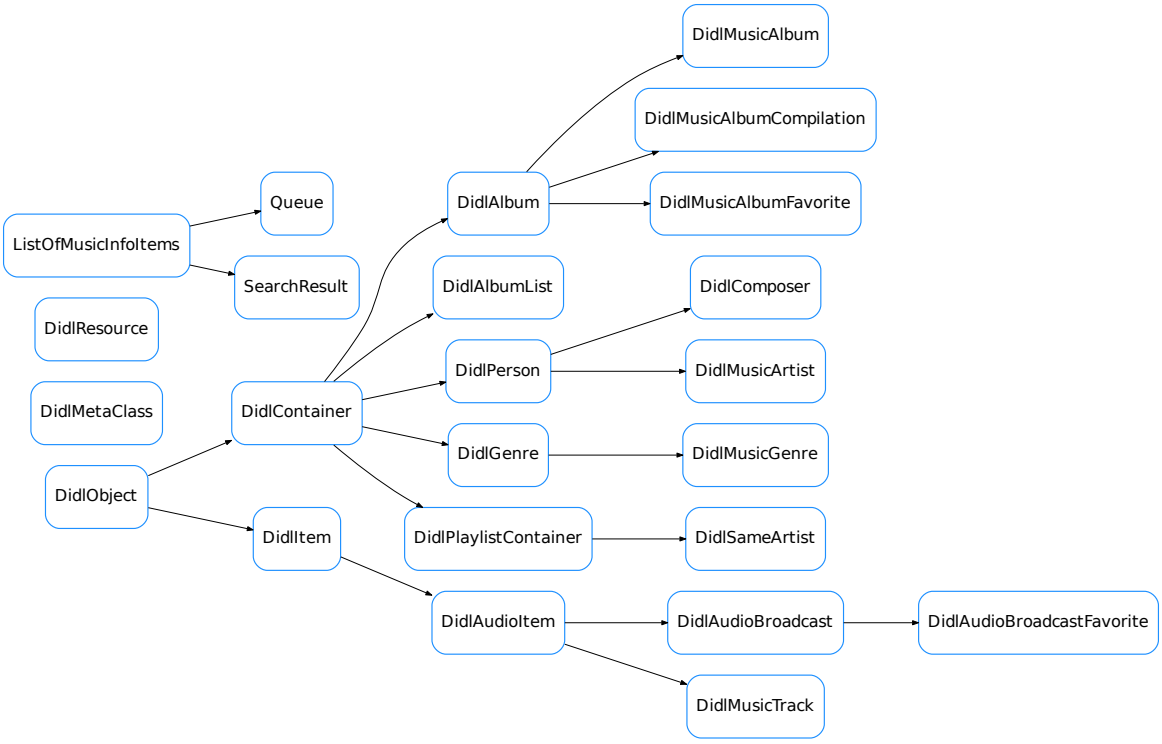
1.8 The Music Library Data Structures

This page contains a thorough introduction to the data structures used for the music library items[1]. The data structures are implemented in the *soco.data_structures* module and they are used to represent the metadata for music items, such as music tracks, albums, genres and playlists.

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 therefore possible and useful to derive a hierarchy of items, and to implement them as a class hierarchy. 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 left to right.

[1] Text of the first footnote.



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`.

1.9 soco package

1.9.1 Subpackages

soco.music_services package

Submodules

soco.music_services.accounts module This module contains classes relating to Third Party music services.

class `soco.music_services.accounts.`**Account**
An account for a Music Service.

Each service may have more than one account: see the Sonos release notes for version 5-2

service_type = None
str: A unique identifier for the music service to which this account relates, eg `'2311'` for Spotify.

serial_number = None
str: A unique identifier for this account

nickname = None
str: The account's nickname

deleted = None
bool: `True` if this account has been deleted

username = None
str: The username used for logging into the music service

metadata = None
str: Metadata for the account

oa_device_id = None
str: Used for OpenAuth id for some services

key = None
str: Used for OpenAuthid for some services

classmethod get_accounts(*soco=None*)
Get all accounts known to the Sonos system.

Parameters **soco** (*SoCo*, optional) – a *SoCo* instance to query. If `None`, a random instance is used. Defaults to `None`.

Returns A dict containing account instances. Each key is the account's serial number, and each value is the related Account instance. Accounts which have been marked as deleted are excluded.

Return type dict

Note: Any existing Account instance will have its attributes updated to those currently stored on the Sonos system.

---

classmethod get_accounts_for_service(*service_type*)
Get a list of accounts for a given music service.

Parameters **service_type** (*str*) – The service_type to use.

Returns A list of *Account* instances.

Return type list

soco.music_services.music_service module Sonos Music Services interface.

This module provides the MusicService class and related functionality.

class soco.music_services.music_service.**MusicServiceSoapClient**(*endpoint*, *timeout*, *music_service*)
A SOAP client for accessing Music Services.

This class handles all the necessary authentication for accessing third party music services. You are unlikely to need to use it yourself.

Parameters

- **endpoint** (*str*) – The SOAP endpoint. A url.
- **timeout** (*int*) – Timeout the connection after this number of seconds.
- **music_service** (MusicService) – The MusicService object to which this client belongs.

get_soap_header()
Generate the SOAP authentication header for the related service.

This header contains all the necessary authentication details.

Returns A string representation of the XML content of the SOAP header.

Return type str

call(*method*, *args=None*)
Call a method on the server.

Parameters

- **method** (*str*) – The name of the method to call.
- **args** (*List[Tuple[str, str]] or None*) – A list of (parameter, value) pairs representing the parameters of the method. Defaults to None.

Returns An OrderedDict representing the response.

Return type OrderedDict

Raises *MusicServiceException* – containing details of the error returned by the music service.

class soco.music_services.music_service.**MusicService**(*service_name*, *account=None*)
The MusicService class provides access to third party music services.

Example

List all the services Sonos knows about:

```
>>> from soco.music_services import MusicService
>>> print(MusicService.get_all_music_services_names())
['Spotify', 'The Hype Machine', 'Saavn', 'Bandcamp',
 'Stitcher SmartRadio', 'Concert Vault',
 ...
 ]
```

Or just those to which you are subscribed:

```
>>> print(MusicService.get_subscribed_services_names())
['Spotify', 'radioPup', 'Spreaker']
```

Interact with TuneIn:

```
>>> tunein = MusicService('TuneIn')
>>> print (tunein)
<MusicService 'TuneIn' at 0x10ad84e10>
```

Browse an item. By default, the root item is used. An OrderedDict is returned:

```
>>> from json import dumps # Used for pretty printing ordereddicts
>>> print(dumps(tunein.get_metadata(), indent=4)
{
```

```
        "index": "0",
        "count": "7",
        "total": "7",
        "mediaCollection": [
            {
                "id": "featured:c100000150",
                "title": "Blue Note on SONOS",
                "itemType": "container",
                "authRequired": "false",
                "canPlay": "false",
                "canEnumerate": "true",
                "canCache": "true",
                "homogeneous": "false",
                "canAddToFavorite": "false",
                "canScroll": "false",
                "albumArtURI":
                "http://cdn-albums.tunein.com/sonos/channel_legacy.png"
            },
            {
                "id": "y1",
                "title": "Music",
                "itemType": "container",
                "authRequired": "false",
                "canPlay": "false",
                "canEnumerate": "true",
                "canCache": "true",
                "homogeneous": "false",
                "canAddToFavorite": "false",
                "canScroll": "false",
                "albumArtURI": "http://cdn-albums.tunein.com/sonos...
                .png"
            },
     ...
        ]
    }
```

Interact with Spotify (assuming you are subscribed):

```
>>> spotify = MusicService('Spotify')
```

Get some metadata about a specific track:

```
>>> response =  spotify.get_media_metadata(
... item_id='spotify:track:6NmXV4o6bmp704aPGyTVVG')
>>> print(dumps(response, indent=4)
{
    "mediaMetadata": {
        "id": "spotify:track:6NmXV4o6bmp704aPGyTVVG",
        "itemType": "track",
        "title": "Bøn Fra Helvete (Live)",
        "mimeType": "audio/x-spotify",
        "trackMetadata": {
            "artistId": "spotify:artist:1s1DnVoBDfp3jxjjew8cBR",
            "artist": "Kaizers Orchestra",
            "albumId": "spotify:album:6K8NUknbPh5TGaKeZdDwSg",
            "album": "Mann Mot Mann (Ep)",
            "duration": "317",
            "albumArtURI":
            "http://o.scdn.co/image/7b76a5074416e83fa3f3cd...9",
```

```
                "canPlay": "true",
                "canSkip": "true",
                "canAddToFavorites": "true"
            }
        }
    }
    or even a playlist:
```

```
    >>> response =  spotify.get_metadata(
    ...     item_id='spotify:user:spotify:playlist:0FQk6BADgIIYd3yTLCThjg')
```

Find the available search categories, and use them:

```
    >>> print(spotify.available_search_categories)
    ['albums', 'tracks', 'artists']
    >>> result =  spotify.search(category='artists', term='miles')
```

Note: Some of this code is still unstable, and in particular the data structures returned by methods such as *get_metadata* may change in future.

Parameters

- **service_name** (*str*) – The name of the music service, as returned by *get_all_music_services_names()*, eg 'Spotify', or 'TuneIn'
- **account** (Account) – The account to use to access this service. If none is specified, one will be chosen automatically if possible. Defaults to None.

Raises *MusicServiceException*

classmethod get_all_music_services_names()
Get a list of the names of all available music services.

These services have not necessarily been subscribed to.

Returns A list of strings.

Return type list

classmethod get_subscribed_services_names()
Get a list of the names of all subscribed music services.

Returns A list of strings.

Return type list

classmethod get_data_for_name(*service_name*)
Get the data relating to a named music service.

Parameters **service_name** (*str*) – The name of the music service for which data is required.

Returns Data relating to the music service.

Return type dict

Raises *MusicServiceException* – if the music service cannot be found.

available_search_categories
list: The list of search categories (each a string) supported.

May include 'artists', 'albums', 'tracks', 'playlists', 'genres', 'stations', 'tags', or others depending on the service. Some services, such as Spotify, support 'all', but do not advertise it.

Any of the categories in this list may be used as a value for category in *search()*.

Example

```
>>> print(spotify.available_search_categories)
['albums', 'tracks', 'artists']
>>> result =  spotify.search(category='artists', term='miles')
```

sonos_uri_from_id(*item_id*)

Get a uri which can be sent for playing.

Parameters **item_id** (*str*) – The unique id of a playable item for this music service, such as that returned in the metadata from *get_metadata*, eg spotify:track:2qs5ZcLByNTctJKbhAZ9JE

Returns A URI of the form: soco://spotify%3Atrack%3A2qs5ZcLByNTctJKbhAZ9JE?sid=2311&sn=1 which encodes the item_id, and relevant data from the account for the music service. This URI can be sent to a Sonos device for playing, and the device itself will retrieve all the necessary metadata such as title, album etc.

Return type str

desc

str: The Sonos descriptor to use for this service.

The Sonos descriptor is used as the content of the <desc> tag in DIDL metadata, to indicate the relevant music service id and username.

get_metadata(*item_id='root'*, *index=0*, *count=100*, *recursive=False*)

Get metadata for a container or item.

Parameters

- **item_id** (*str*) – The container or item to browse. Defaults to the root item.
- **index** (*int*) – The starting index. Default 0.
- **count** (*int*) – The maximum number of items to return. Default 100.
- **recursive** (*bool*) – Whether the browse should recurse into sub-items (Does not always work). Defaults to False.

Returns The item or container's metadata, or None.

Return type OrderedDict

See also:

The Sonos getMetadata API.

search(*category*, *term=''*, *index=0*, *count=100*)

Search for an item in a category.

Parameters

- **category** (*str*) – The search category to use. Standard Sonos search categories are 'artists', 'albums', 'tracks', 'playlists', 'genres', 'stations', 'tags'. Not all are available for each music service. Call available_search_categories for a list for this service.

- **term** (*str*) – The term to search for.
- **index** (*int*) – The starting index. Default 0.
- **count** (*int*) – The maximum number of items to return. Default 100.

Returns The search results, or None.

Return type OrderedDict

See also:

The Sonos search API

get_media_metadata(*item_id*)
Get metadata for a media item.

Parameters **item_id** (*str*) – The item for which metadata is required.

Returns The item's metadata, or None

Return type OrderedDict

See also:

The Sonos getMediaMetadata API

get_media_uri(*item_id*)
Get a streaming URI for an item.

Note: You should not need to use this directly. It is used by the Sonos players (not the controllers) to obtain the uri of the media stream. If you want to have a player play a media item, you should add add it to the queue using its id and let the player work out where to get the stream from (see On Demand Playback and Programmed Radio)

Parameters **item_id** (*str*) – The item for which the URI is required

Returns The item's streaming URI.

Return type str

get_last_update()
Get last_update details for this music service.

Returns A dict with keys 'catalog', and 'favorites'. The value of each is a string which changes each time the catalog or favorites change. You can use this to detect when any caches need to be updated.

Return type OrderedDict

get_extended_metadata(*item_id*)
Get extended metadata for a media item, such as related items.

Parameters **item_id** (*str*) – The item for which metadata is required.

Returns The item's extended metadata or None.

Return type OrderedDict

See also:

The Sonos getExtendedMetadata API

get_extended_metadata_text(*item_id*, *metadata_type*)
Get extended metadata text for a media item.

Parameters

- **item_id** (*str*) – The item for which metadata is required
- **metadata_type** (*str*) – The type of text to return, eg
- **or 'ALBUM_NOTES'. Calling**(*'ARTIST_BIO',*) –
- **for the item will show which extended** (get_extended_metadata) –
- **are available** (*metadata_types*) –

Returns The item's extended metadata text or None

Return type str

See also:

The Sonos getExtendedMetadataText API

soco.music_services.music_service.**desc_from_uri**(*uri*)
Create the content of DIDL desc element from a uri.

Parameters **uri** (*str*) – A uri, eg: 'x-sonos-http:track%3a3402413.mp3?sid=2&flags=32&s

Returns The content of a desc element for that uri, eg 'SA_RINCON519_email@example.com'

Return type str

soco.plugins package

Submodules

soco.plugins.example module Example implementation of a plugin.

class soco.plugins.example.**ExamplePlugin**(*soco*, *username*)
This file serves as an example of a SoCo plugin.

Initialize the plugin.

The plugin can accept any arguments it requires. It should at least accept a soco instance which it passes on to the base class when calling super's __init__.

music_plugin_play()
Play some music.

This is just a reimplementation of the ordinary play function, to show how we can use the general upnp methods from soco

music_plugin_stop()
Stop the music.

This methods shows how, if we need it, we can use the soco functionality from inside the plugins

soco.plugins.spotify module The Spotify plugin has been DEPRECATED

The Spotify Plugin has been immediately deprecated (August 2016), because the API it was based on (The Spotify Metadata API) has been ended. Since this rendered the plug-in broken, there was no need to forewarn of the deprecation.

Please consider moving to the new general music services code (in soco.music_services.music_service), that makes it possible to retrived information about the available media from all music services. A short intro for how to use the new code is available in the API documentation:

- http://docs.python-soco.com/en/latest/api/soco.music_services.music_service.html

and for some information about how to add items from the music services to the queue, see this gist:

- https://gist.github.com/lawrenceakka/2d21dca590b4fa7e3af2"

This deprecation notification will be deleted for the second release after 0.12.

soco.plugins.wimp module Plugin for the Wimp music service (Service ID 20)

class soco.plugins.wimp.**Wimp**(*soco*, *username*, *retries=3*, *timeout=3.0*)
Class that implements a Wimp plugin.

Note: There is an (apparent) in-consistency in the use of one data type from the Wimp service. When searching for playlists, the XML returned by the Wimp server indicates, that the type is an 'album list', and it thus suggest, that this type is used for a list of tracks (as expected for a playlist), and this data type is reported to be playable. However, when browsing the music tree, the Wimp server will return items of 'album list' type, but in this case it is used for a list of albums and it is not playable. This plugin maintains this (apparent) in-consistency to stick as close to the reported data as possible, so search for playlists returns MSAlbumList that are playable and while browsing the content tree the MSAlbumList items returned to you are not playable.

Note: Wimp in some cases lists tracks that are not available. In these cases, while it will correctly report these tracks as not being playable, the containing data structure like e.g. the album they are on may report that they are playable. Trying to add one of these to the queue will return a SoCoUPnPException with error code '802'.

Initialize the plugin.

Parameters

- **soco** – The soco instance to retrieve the session ID for the music service
- **username** (*str*) – The username for the music service
- **retries** (*int*) – The number of times to retry before giving up
- **timeout** (*float*) – The time to wait for the post to complete, before timing out. The Wimp server seems either slow to respond or to make the queries internally, so the timeout should probably not be shorter than 3 seconds.

Type soco.SoCo

Note: If you are using a phone number as the username and are experiencing problems connecting, then try to prepend the area code (no + or 00). I.e. if your phone number is 12345678 and you are from denmark, then use 4512345678. This must be set up the same way in the Sonos device. For details see here (In Danish)

---

name
Return the human read-able name for the plugin

username
Return the username.

service_id
Return the service id.

description
Return the music service description for the DIDL metadata on the form `'SA_RINCON5127_...self.username...'`

get_tracks(*search*, *start=0*, *max_items=100*)
Search for tracks.

See get_music_service_information for details on the arguments

get_albums(*search*, *start=0*, *max_items=100*)
Search for albums.

See get_music_service_information for details on the arguments

get_artists(*search*, *start=0*, *max_items=100*)
Search for artists.

See get_music_service_information for details on the arguments

get_playlists(*search*, *start=0*, *max_items=100*)
Search for playlists.

See get_music_service_information for details on the arguments.

Note: Un-intuitively this method returns MSAlbumList items. See note in class doc string for details.

---

get_music_service_information(*search_type*, *search*, *start=0*, *max_items=100*)
Search for music service information items.

Parameters

- **search_type** (*str*) – The type of search to perform, possible values are: 'artists', 'albums', 'tracks' and 'playlists'
- **search** (*str*) – The search string to use
- **start** (*int*) – The starting index of the returned items
- **max_items** (*int*) – The maximum number of returned items

---

Note: Un-intuitively the playlist search returns MSAlbumList items. See note in class doc string for details.

---

browse(*ms_item=None*)
Return the sub-elements of item or of the root if item is None

Parameters **item** – Instance of sub-class of `soco.data_structures.MusicServiceItem`. This object must have item_id, service_id and extended_id properties

---

Note: Browsing a MSTrack item will return itself.

Note: This plugin cannot yet set the parent ID of the results correctly when browsing `soco.data_structures.MSFavorites` and `soco.data_structures.MSCollection` elements.

static id_to_extended_id(*item_id*, *item_class*)
Return the extended ID from an ID.

Parameters

- **item_id** (*str*) – The ID of the music library item
- **cls** (Sub-class of `soco.data_structures.MusicServiceItem`) – The class of the music service item

The extended id can be something like 00030020trackid_22757082 where the id is just trackid_22757082. For classes where the prefix is not known returns None.

static form_uri(*item_content*, *item_class*)
Form the URI for a music service element.

Parameters

- **item_content** (*dict*) – The content dict of the item
- **item_class** (Sub-class of `soco.data_structures.MusicServiceItem`) – The class of the item

Module contents

This is the __init__ module for the plugins.

It contains the base class for all plugins

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

name
human-readable name of the plugin

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

1.9.2 Submodules

soco.alarms module

This module contains classes relating to Sonos Alarms.

`soco.alarms.`**is_valid_recurrence**(*text*)
Check that `text` is a valid recurrence string.

A valid recurrence string is `DAILY`, `ONCE`, `WEEKDAYS`, `WEEKENDS` or of the form `ON_DDDDDD` where `D` is a number from 0-7 representing a day of the week (Sunday is 0), e.g. `ON_034` meaning Sunday, Wednesday and Thursday

Parameters **text** (*str*) – the recurrence string to check.

Returns `True` if the recurrence string is valid, else `False`.

Return type bool

Examples

```
>>> from soco.alarms import is_valid_recurrence
>>> is_valid_recurrence('WEEKENDS')
True
>>> is_valid_recurrence('')
False
>>> is_valid_recurrence('ON_132')  # Mon, Tue, Wed
True
>>> is_valid_recurrence('ON_777')  # Sat
True
>>> is_valid_recurrence('ON_3421') # Mon, Tue, Wed, Thur
True
>>> is_valid_recurrence('ON_123456789') # Too many digits
False
```

class soco.alarms.**Alarm**(*zone*, *start_time=None*, *duration=None*, *recurrence='DAILY'*, *enabled=True*, *program_uri=None*, *program_metadata=''*, *play_mode='NORMAL'*, *volume=20*, *include_linked_zones=False*)

A class representing a Sonos Alarm.

Alarms may be created or updated and saved to, or removed from the Sonos system. An alarm is not automatically saved. Call *save()* to do that.

Example

```
>>> device = discovery.any_soco()
>>> # create an alarm with default properties
>>> alarm = Alarm(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([])
```

Parameters

- **zone** (*SoCo*) – The soco instance which will play the alarm.
- **start_time** (datetime.time, optional) – The alarm's start time. Specify hours, minutes and seconds only. Defaults to the current time.
- **duration** (datetime.time, optional) – The alarm's duration. Specify hours, minutes and seconds only. May be None for unlimited duration. Defaults to None.

- **recurrence** (*str, optional*) – A string representing how often the alarm should be triggered. Can be DAILY, ONCE, WEEKDAYS, WEEKENDS or of the form ON_DDDDDD where D is a number from 0-7 representing a day of the week (Sunday is 0), e.g. ON_034 meaning Sunday, Wednesday and Thursday. Defaults to DAILY.
- **enabled** (*bool, optional*) – True if alarm is enabled, False otherwise. Defaults to True.
- **program_uri** (*str, optional*) – The uri to play. If None, the built-in Sonos chime sound will be used. Defaults to None.
- **program_metadata** (*str, optional*) – The metadata associated with *program_uri*. Defaults to ''.
- **play_mode** (*str, optional*) – The play mode for the alarm. Can be one of NORMAL, SHUFFLE_NOREPEAT, SHUFFLE, REPEAT_ALL, REPEAT_ONE, SHUFFLE_REPEAT_ONE. Defaults to NORMAL.
- **volume** (*int, optional*) – The alarm's volume (0-100). Defaults to 20.
- **include_linked_zones** (*bool, optional*) – True if the alarm should be played on the other speakers in the same group, False otherwise. Defaults to False.

start_time = None
datetime.time: The alarm's start time.

duration = None
datetime.time: The alarm's duration.

enabled = None
bool: True if the alarm is enabled, else False.

program_uri = None

program_metadata = None
str: The uri to play.

include_linked_zones = None
bool: True if the alarm should be played on the other speakers in the same group, False otherwise.

play_mode
str: The play mode for the alarm.

Can be one of NORMAL, SHUFFLE_NOREPEAT, SHUFFLE, REPEAT_ALL, REPEAT_ONE, SHUFFLE_REPEAT_ONE.

volume
int: The alarm's volume (0-100).

recurrence
str: How often the alarm should be triggered.

Can be DAILY, ONCE, WEEKDAYS, WEEKENDS or of the form ON_DDDDDDD where D is a number from 0-7 representing a day of the week (Sunday is 0), e.g. ON_034 meaning Sunday, Wednesday and Thursday.

save()
Save the alarm to the Sonos system.

Raises *SoCoUPnPException* – if the alarm cannot be created because there is already an alarm for this room at the specified time.

remove()
Remove the alarm from the Sonos system.

There is no need to call *save*. The Python instance is not deleted, and can be saved back to Sonos again if desired.

soco.alarms.**get_alarms**(*zone=None*)
Get a set of all alarms known to the Sonos system.

Parameters **zone** (*SoCo*, optional) – a SoCo instance to query. If None, a random instance is used. Defaults to None.

Returns A set of *Alarm* instances

Return type set

Note: Any existing *Alarm* instance will have its attributes updated to those currently stored on the Sonos system.

soco.cache module

This module contains the classes underlying SoCo's caching system.

class soco.cache.**_BaseCache**(**args*, ***kwargs*)
An abstract base class for the cache.

enabled = None
bool: whether the cache is enabled

put(*item*, **args*, ***kwargs*)
Put an item into the cache.

get(**args*, ***kwargs*)
Get an item from the cache.

delete(**args*, ***kwargs*)
Delete an item from the cache.

clear()
Empty the whole cache.

class soco.cache.**NullCache**(**args*, ***kwargs*)
A cache which does nothing.

Useful for debugging.

put(*item*, **args*, ***kwargs*)
Put an item into the cache.

get(**args*, ***kwargs*)
Get an item from the cache.

delete(**args*, ***kwargs*)
Delete an item from the cache.

clear()
Empty the whole cache.

class soco.cache.**TimedCache**(*default_timeout=0*)
A simple thread-safe cache for caching method return values.

The cache key is generated by from the given `*args` and `**kwargs`. Items are expired from the cache after a given period of time.

Example

```
>>> from time import sleep
>>> cache = TimedCache()
>>> cache.put("item", 'some', kw='args', timeout=3)
>>> # Fetch the item again, by providing the same args and kwargs.
>>> assert cache.get('some', kw='args') == "item"
>>> # Providing different args or kwargs will not return the item.
>>> assert not cache.get('some', 'otherargs') == "item"
>>> # Waiting for less than the provided timeout does not cause the
>>> # item to expire.
>>> sleep(2)
>>> assert cache.get('some', kw='args') == "item"
>>> # But waiting for longer does.
>>> sleep(2)
>>> assert not cache.get('some', kw='args') == "item"
```

Warning: At present, the cache can theoretically grow and grow, since entries are not automatically purged, though in practice this is unlikely since there are not that many different combinations of arguments in the places where it is used in SoCo, so not that many different cache entries will be created. If this becomes a problem, use a thread and timer to purge the cache, or rewrite this to use LRU logic!

Parameters

- **default_timeout** (*int*) – The default number of seconds after
- **items will be expired.** (*which*) –

default_timeout = None
`int`: The default caching expiry interval in seconds.

get(**args*, ***kwargs*)
Get an item from the cache for this combination of args and kwargs.

Parameters

- ***args** – any arguments.
- ****kwargs** – any keyword arguments.

Returns The object which has been found in the cache, or `None` if no unexpired item is found. This means that there is no point storing an item in the cache if it is `None`.

Return type object

put(*item*, **args*, ***kwargs*)
Put an item into the cache, for this combination of args and kwargs.

Parameters

- ***args** – any arguments.
- ****kwargs** – any keyword arguments. If `timeout` is specified as one of the keyword arguments, the item will remain available for retrieval for `timeout` seconds. If `timeout` is `None` or not specified, the `default_timeout` for this cache will be used. Specify a `timeout` of 0 (or ensure that the `default_timeout` for this cache is 0) if this item is not to be cached.

delete(**args*, ***kwargs*)
Delete an item from the cache for this combination of args and kwargs.

clear()
Empty the whole cache.

static make_key(**args*, ***kwargs*)
Generate a unique, hashable, representation of the args and kwargs.

Parameters

- ***args** – any arguments.
- ****kwargs** – any keyword arguments.

Returns the key.

Return type str

class soco.cache.**Cache**(**args*, ***kwargs*)
A factory class which returns an instance of a cache subclass.

A *TimedCache* is returned, unless *config.CACHE_ENABLED* is False, in which case a *NullCache* will be returned.

clear()
Empty the whole cache.

delete(**args*, ***kwargs*)
Delete an item from the cache.

get(**args*, ***kwargs*)
Get an item from the cache.

put(*item*, **args*, ***kwargs*)
Put an item into the cache.

soco.compat module

This module contains various compatibility definitions and imports.

It is used internally by SoCo to ensure compatibility with Python 2.

soco.compat.**with_metaclass**(*meta*, **bases*)
A Python 2/3 compatible way of declaring a metaclass.

Taken from Jinja 2 via python-future. License: BSD. Use it like this:

```
class MyClass(with_metaclass(MyMetaClass, BaseClass)):
        pass
```

soco.config module

This module contains configuration variables.

They may be set by your code as follows:

```
from soco import config
...
config.VARIABLE = value
```

soco.config.**SOCO_CLASS**
The class object to use when *SoCo* instances are created.

Specify the actual callable class object here, not a string. If `None`, the default SoCo class will be used. Must be set before any instances are created, or it will have unpredictable effects.

alias of `SoCo`

`soco.config.`**`CACHE_ENABLED`** = **True**
Is the cache enabled?

If `True` (the default), some caching of network requests will take place.

See also:

The *soco.cache* module.

`soco.config.`**`EVENT_LISTENER_PORT`** = **1400**
The port on which the event listener listens.

The default is 1400. You must set this before subscribing to any events.

See also:

The *soco.events* module.

soco.core module

The core module contains the SoCo class that implements the main entry to the SoCo functionality

`soco.core.`**`only_on_master`**(*function*)
Decorator that raises SoCoSlaveException on master call on slave.

class `soco.core.`**`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.

Methods

play()	Play the currently selected track.
play_uri([uri, meta, title, start])	Play a given stream.
play_from_queue(index[, start])	Play a track from the queue by index.
pause()	Pause the currently playing track.
stop()	Stop the currently playing track.
seek(timestamp)	Seeks to a given timestamp in the current track, specified in the format
next()	Go to the next track.
previous()	Go back to the previously played 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([refresh, timeout])	Get information about the Sonos speaker.
partymode()	Put all the speakers in the network in the same group, a.k.a Party Mode
join(master)	Join this speaker to another "master" speaker.
unjoin()	Remove this speaker from a group.
get_queue([start, max_items, full_album_art_uri])	Get information about the queue.

Table 1.1 – continued from previous page

get_current_transport_info()	Get the current playback state.
add_uri_to_queue(uri)	Adds the URI to the queue.
add_to_queue(queueable_item)	Adds a queueable item to the queue.
remove_from_queue(index)	Remove a track from the queue by index.
clear_queue()	Removes all tracks from the queue.
get_favorite_radio_shows([start, max_items])	Get favorite radio shows from Sonos' Radio app.
get_favorite_radio_stations([start, max_items])	Get favorite radio stations from Sonos' Radio app.
get_sonos_favorites([start, max_items])	Get Sonos favorites.
create_sonos_playlist(title)	Create a new empty Sonos playlist.
create_sonos_playlist_from_queue(title)	Create a new Sonos playlist from the current queue.
remove_sonos_playlist(sonos_playlist)	Remove a Sonos playlist.
add_item_to_sonos_playlist(queueable_item, ...)	Adds a queueable item to a Sonos' playlist.
get_item_album_art_uri(item)	Get an item's Album Art absolute URI.
set_sleep_timer(sleep_time_seconds)	Sets the sleep timer.
get_sleep_timer()	Retrieves remaining sleep time, if any

Properties

Warning: These properties are not generally cached and may 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.

uid	A unique identifier.
household_id	A unique identifier for all players in a household.
mute	The speaker's mute state.
volume	The speaker's volume.
bass	The speaker's bass EQ.
treble	The speaker's treble EQ.
loudness	The Sonos speaker's loudness compensation.
cross_fade	The speaker's cross fade state.
status_light	The white Sonos status light between the mute button and the volume up button on the speaker.
player_name	The speaker's name.
play_mode	str: The queue's play mode.
queue_size	Get size of queue.
is_playing_tv	Is the playbar speaker input from TV?
is_playing_radio	Is the speaker playing radio?
is_playing_line_in	Is the speaker playing line-in?

ip_address = None
The speaker's ip address

player_name
The speaker's name.

A string.

uid
A unique identifier.

Looks like: `'RINCON_000XXXXXXXXXX1400'`

household_id
A unique identifier for all players in a household.

Looks like: `'Sonos_asahHKgjgJGjgjGjggjJgjJG34'`

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

is_bridge
Is this zone a bridge?

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

return True or False

play_mode
str: 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.

cross_fade
The speaker's cross fade state.

True if enabled, False otherwise

play_from_queue(*index*, *start=True*)
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()
Play the currently selected track.

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

Raises SoCoException (or a subclass) upon errors.

play_uri(*uri=''*, *meta=''*, *title=''*, *start=True*)
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.

pause()
Pause the currently playing track.

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

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.

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.

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).

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.

mute
The speaker's mute state.

True if muted, False otherwise

volume
The speaker's volume.

An integer between 0 and 100.

bass
The speaker's bass EQ.

An integer between -10 and 10.

treble
The speaker's treble EQ.

An integer between -10 and 10.

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

all_groups
Return a set of all the available groups.

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.

all_zones
Return a set of all the available zones.

visible_zones
Return an set of all visible zones.

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.

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

---

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.

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.

is_playing_radio
Is the speaker playing radio?

return True or False

is_playing_line_in
Is the speaker playing line-in?

return True or False

is_playing_tv
Is the playbar speaker input from TV?

return True or False

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.

status_light
The white Sonos status light between the mute button and the volume up button on the speaker.

True if on, otherwise False.

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.

Note: Calling this method on a slave in a group will not return the track the group is playing, but the last track this speaker was playing.

---

get_speaker_info(*refresh=False*, *timeout=None*)
Get information about the Sonos speaker.

Arguments: refresh – Refresh the speaker info cache. timeout – How long to wait for the server to send

Unexpected indentation.

> data before giving up, as a float, or a (**'connect timeout, read timeout'_**) tuple e.g. (3, 5). Default is no timeout.

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

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_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

queue_size
Get size of queue.

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

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

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

add_to_queue(*queueable_item*)
Adds a queueable item to the queue.

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.

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_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_sonos_favorites(*start=0*, *max_items=100*)
Get Sonos favorites.

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*, *uri* and `meta` 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.

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(*title*)
Create a new Sonos playlist from the current queue.

Params title Name of the playlist

Returns An instance of *`DidlPlaylistContainer`*

remove_sonos_playlist(*sonos_playlist*)
Remove a Sonos playlist.

Parameters **sonos_playlist** (`DidlPlaylistContainer`) – Sonos playlist to remove or the item_id (str).

Returns True if succesful, False otherwise

Return type bool

Raises `SoCoUPnPException` – If sonos_playlist does not point to a valid object.

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

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

set_sleep_timer(*sleep_time_seconds*)
Sets the sleep timer.

Parameters **sleep_time_seconds** (*`int or NoneType`*) – How long to wait before turning off speaker in seconds, None to cancel a sleep timer. Maximum value of 86399

Raises

- `SoCoException` – Upon errors interacting with Sonos controller
- `ValueError` – Argument/Syntax errors

get_sleep_timer()
Retrieves remaining sleep time, if any

Returns Number of seconds left in timer. If there is no sleep timer currently set it will return None.

Return type int or NoneType

Raises SoCoException (or a subclass) upon errors.

get_artists(**args*, ***kwargs*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_artists` instead.

get_album_artists(**args*, ***kwargs*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_album_artists` instead.

get_albums(**args*, ***kwargs*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_music_library_information` instead.

get_genres(**args*, ***kwargs*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_music_library_information` instead.

get_composers(**args*, ***kwargs*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_composers` instead.

get_tracks(**args*, ***kwargs*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_tracks` instead.

get_playlists(**args*, ***kwargs*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_playlists` instead.

get_music_library_information(*search_type*, *start=0*, *max_items=100*, *full_album_art_uri=False*, *search_term=None*, *subcategories=None*, *complete_result=False*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_music_library_information` instead.

browse(*ml_item=None*, *start=0*, *max_items=100*, *full_album_art_uri=False*, *search_term=None*, *subcategories=None*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.browse` instead.

browse_by_idstring(*search_type*, *idstring*, *start=0*, *max_items=100*, *full_album_art_uri=False*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.browse_by_idstring` instead.

library_updating
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.library_updating` instead.

start_library_update(*album_artist_display_option=''*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.start_library_update` instead.

search_track(*artist*, *album=None*, *track=None*, *full_album_art_uri=False*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.search_track` instead.

get_albums_for_artist(*artist*, *full_album_art_uri=False*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_albums_for_artist` instead.

get_tracks_for_album(*artist*, *album*, *full_album_art_uri=False*)
Deprecated since version 0.12: Will be removed in version 0.14. Use `soco.music_library.get_tracks_for_album` instead.

album_artist_display_option
Deprecated since version 0.12: Will be removed in version 0.14. Use

soco.music_library.album_artist_display instead.

reorder_sonos_playlist(*sonos_playlist*, *tracks*, *new_pos*, *update_id=0*)

Reorder and/or Remove tracks in a Sonos playlist.

The underlying call is quite complex as it can both move a track within the list or delete a track from the playlist. All of this depends on what tracks and new_pos specify.

If a list is specified for tracks, then a list must be used for new_pos. Each list element is a discrete modification and the next list operation must anticipate the new state of the playlist.

If a comma formatted string to tracks is specified, then use a similiar string to specify new_pos. Those operations should be ordered from the end of the list to the beginning

See the helper methods *clear_sonos_playlist()*, *move_in_sonos_playlist()*, *remove_from_sonos_playlist()* for simplified usage.

update_id - If you have a series of operations, tracking the update_id and setting it, will save a lookup operation.

Examples

To reorder the first two tracks:

```
# sonos_playlist specified by the DidlPlaylistContainer object
sonos_playlist = device.get_sonos_playlists()[0]
device.reorder_sonos_playlist(sonos_playlist,
                              tracks=[0, ], new_pos=[1, ])
# OR specified by the item_id
device.reorder_sonos_playlist('SQ:0', tracks=[0, ], new_pos=[1, ])
```

To delete the second track:

```
# tracks/new_pos are a list of int
device.reorder_sonos_playlist(sonos_playlist,
                              tracks=[1, ], new_pos=[None, ])
# OR tracks/new_pos are a list of int-like
device.reorder_sonos_playlist(sonos_playlist,
                              tracks=['1', ], new_pos=['', ])
# OR tracks/new_pos are strings - no transform is done
device.reorder_sonos_playlist(sonos_playlist, tracks='1',
                              new_pos='')
```

To reverse the order of a playlist with 4 items:

```
device.reorder_sonos_playlist(sonos_playlist, tracks='3,2,1,0',
                              new_pos='0,1,2,3')
```

Parameters

- **sonos_playlist** – (*DidlPlaylistContainer*): The Sonos playlist object or the item_id (str) of the Sonos playlist.
- **tracks** – (list): list of track indices(int) to reorder. May also be a list of int like things. i.e. `['0', '1',]` OR it may be a str of comma separated int like things. `"0,1"`. Tracks are **0**-based. Meaning the first track is track 0, just like indexing into a Python list.
- **new_pos** (*list*) – list of new positions (int|None) corresponding to track_list. MUST be the same type as `tracks`. **0**-based, see tracks above. `None` is the indicator to remove the track. If using a list of strings, then a remove is indicated by an empty string.

- **update_id** (*int*) – operation id (default: 0) If set to 0, a lookup is done to find the correct value.

Returns Which contains 3 elements: change, length and update_id. Change in size between original playlist and the resulting playlist, the length of resulting playlist, and the new update_id.

Return type dict

Raises SoCoUPnPException – If playlist does not exist or if your tracks and/or new_pos arguments are invalid.

clear_sonos_playlist(*sonos_playlist*, *update_id=0*)

Clear all tracks from a Sonos playlist. This is a convenience method for *reorder_sonos_playlist()*.

Example:

```
device.clear_sonos_playlist(sonos_playlist)
```

Parameters

- **sonos_playlist** – (*DidlPlaylistContainer*): Sonos playlist object or the item_id (str) of the Sonos playlist.
- **update_id** (*int*) – Optional update counter for the object. If left at the default of 0, it will be looked up.

Returns See *reorder_sonos_playlist()*

Return type dict

Raises

- ValueError – If sonos_playlist specified by string and is not found.
- SoCoUPnPException – See *reorder_sonos_playlist()*

move_in_sonos_playlist(*sonos_playlist*, *track*, *new_pos*, *update_id=0*)

Move a track to a new position within a Sonos Playlist. This is a convenience method for *reorder_sonos_playlist()*.

Example:

```
device.move_in_sonos_playlist(sonos_playlist, track=0, new_pos=1)
```

Parameters

- **sonos_playlist** – (*DidlPlaylistContainer*): Sonos playlist object or the item_id (str) of the Sonos playlist.
- **track** (*int*) – **0**-based position of the track to move. The first track is track 0, just like indexing into a Python list.
- **new_pos** (*int*) – **0**-based location to move the track.
- **update_id** (*int*) – Optional update counter for the object. If left at the default of 0, it will be looked up.

Returns See *reorder_sonos_playlist()*

Return type dict

Raises SoCoUPnPException – See *reorder_sonos_playlist()*

remove_from_sonos_playlist(*sonos_playlist*, *track*, *update_id=0*)

Remove a track from a Sonos Playlist. This is a convenience method for *reorder_sonos_playlist()*.

Example:

```
device.remove_from_sonos_playlist(sonos_playlist, track=0)
```

Parameters

- **sonos_playlist** – (*DidlPlaylistContainer*): Sonos playlist object or the item_id (str) of the Sonos playlist.
- **track** (*int*) – *0*-based position of the track to move. The first track is track 0, just like indexing into a Python list.
- **update_id** (*int*) – Optional update counter for the object. If left at the default of 0, it will be looked up.

Returns See *reorder_sonos_playlist()*

Return type dict

Raises SoCoUPnPException – See *reorder_sonos_playlist()*

get_sonos_playlist_by_attr(*attr_name*, *match*)

Return the first Sonos Playlist DidlPlaylistContainer that matches the attribute specified.

Parameters

- **attr_name** (*str*) – DidlPlaylistContainer attribute to compare. The most useful being: 'title' and 'item_id'.
- **match** (*str*) – Value to match.

Returns The first matching playlist object.

Return type (*DidlPlaylistContainer*)

Raises

- (AttributeError) – If indicated attribute name does not exist.
- (ValueError) – If a match can not be found.

Example:

```
device.get_sonos_playlist_by_attr('title', 'Foo')
device.get_sonos_playlist_by_attr('item_id', 'SQ:3')
```

soco.data_structures module

This module contains classes for handling DIDL-Lite metadata.

DIDL is the Digital Item Declaration Language , an XML schema which is part of MPEG21. DIDL-Lite is a cut-down version of the schema which is part of the UPnP ContentDirectory specification. It is the XML schema used by Sonos for carrying metadata representing many items such as tracks, playlists, composers, albums etc. Although Sonos uses ContentDirectory v1, the document for v2 [pdf] is more helpful.

soco.data_structures.**to_didl_string**(**args*)

Convert any number of *DidlObjects* to a unicode xml string.

Parameters ***args** (DidlObject) – One or more *DidlObject* (or subclass) instances.

Returns A unicode string representation of DIDL-Lite XML in the form `'<DIDL-Lite ...>...</DIDL-Lite>'`.

Return type str

soco.data_structures.**from_didl_string**(*string*)
Convert a unicode xml string to a list of *DIDLObjects*.

Parameters **string** (*str*) – A unicode string containing an XML representation of one or more DIDL-Lite items (in the form `'<DIDL-Lite ...> ...</DIDL-Lite>'`)

Returns A list of one or more instances of *DidlObject* or a subclass

Return type list

class soco.data_structures.**DidlResource**(*uri*, *protocol_info*, *import_uri=None*, *size=None*, *duration=None*, *bitrate=None*, *sample_frequency=None*, *bits_per_sample=None*, *nr_audio_channels=None*, *resolution=None*, *color_depth=None*, *protection=None*)
Identifies a resource, typically some type of a binary asset, such as a song.

It is represented in XML by a `<res>` element, which contains a uri that identifies the resource.

Parameters

- **uri** (*str*) – value of the `<res>` tag, typically a URI. It **must** be properly escaped (percent encoded) as described in **RFC 3986**
- **protocol_info** (*str*) – a string in the form a:b:c:d that identifies the streaming or transport protocol for transmitting the resource. A value is required. For more information see section 2.5.2 of the UPnP specification [pdf]
- **import_uri** (*str, optional*) – uri locator for resource update.
- **size** (*int, optional*) – size in bytes.
- **duration** (*str, optional*) – duration of the playback of the res at normal speed (`H*:MM:SS:F*` or `H*:MM:SS:F0/F1`)
- **bitrate** (*int, optional*) – bitrate in bytes/second.
- **sample_frequency** (*int, optional*) – sample frequency in Hz.
- **bits_per_sample** (*int, optional*) – bits per sample.
- **nr_audio_channels** (*int, optional*) – number of audio channels.
- **resolution** (*str, optional*) – resolution of the resource (X*Y).
- **color_depth** (*int, optional*) – color depth in bits.
- **protection** (*str, optional*) – statement of protection type.

Note: Not all of the parameters are used by Sonos. In general, only `uri`, `protocol_info` and `duration` seem to be important.

---

uri = None
(str): a percent encoded URI

protocol_info = None
(str): protocol information.

duration = None
str: playback duration

classmethod from_element(*element*)
Set the resource properties from a <res> element.

Parameters **element** (*Element*) – The <res> element

to_element()
Return an ElementTree Element based on this resource.

Returns an Element.

Return type Element

to_dict(*remove_nones=False*)
Return a dict representation of the *DidlResource*.

Parameters **remove_nones** (*bool, optional*) – Optionally remove dictionary elements when their value is None.

Returns a dict representing the *DidlResource*

Return type dict

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

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

Parameters **content** (*dict*) – a dict containing metadata information. Required. Valid keys are the same as the parameters for *DidlResource*.

__eq__(*resource*)
Compare with another DidlResource.

Returns True if all items are equal, else False.

Return type (bool)

class soco.data_structures.**DidlMetaClass**
Meta class for all Didl objects.

static __new__(*mcs*, *name*, *bases*, *attrs*)
Create a new instance.

Parameters

- **name** (*str*) – Name of the class.
- **bases** (*tuple*) – Base classes.
- **attrs** (*dict*) – attributes defined for the class.

class soco.data_structures.**DidlObject**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)
Abstract base class for all DIDL-Lite items.

You should not need to instantiate this. Its XML representation looks like this:

```
<DIDL-Lite xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:r="urn:schemas-rinconnetworks-com:metadata-1-0/"
 xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/">
  <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>
```

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'write_status': ('upnp', 'writeStatus')}
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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.

Parameters **xml** (*Element*) – An `Element` object.

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

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

Parameters **content** (*dict*) – a dict containing metadata information. Required. Valid keys are the same as the parameters for *DidlObject*.

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

Returns `True` if all items are equal, else `False`.

Return type (bool)

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

Returns `True` if any items is unequal, else `False`.

Return type (bool)

__repr__()
Get the repr value for the item.

Returns A string representation of the instance in 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.

Return type str

__str__()
Get the str value for the item.

Returns a string representation in 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.

Return type str

to_dict(*remove_nones=False*)
Return the dict representation of the instance.

Parameters **remove_nones** – Optionally remove dictionary elements when their value is `None`.

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

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

Returns an Element.

Return type Element

class soco.data_structures.**DidlItem**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)
A basic content directory item.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.item'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'radio_show': ('r', 'radioShowMd'), 'creator': ('dc', 'creator'), 'stream_content': ('r', 'streamConte
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

An audio item.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.item.audioItem'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'long_description': ('upnp', 'longDescription'), 'radio_show': ('r', 'radioS
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a music library track.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –

- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.item.audioItem.musicTrack'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'artist': ('upnp', 'artist'), 'creator': ('dc', 'creator'), 'contributor': ('dc', 'contributor'), 'playlist': ('
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents an audio broadcast.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.item.audioItem.audioBroadcast'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'channel_nr': ('upnp', 'channelNr'), 'long_description': ('upnp', 'longDes
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents an audio broadcast Sonos favorite.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.item.audioItem.audioBroadcast.sonos-favorite'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'channel_nr': ('upnp', 'channelNr'), 'long_description': ('upnp', 'longDes
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a music library container.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container'
str - the DIDL Lite class for this object.

tag = 'container'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'write_status': ('upnp', 'writeStatus')}
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

A content directory album.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.album'
str - the DIDL Lite class for this object.

tag = 'container'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'contributor': ('dc', 'contributor'), 'long_description': ('upnp', 'longDesc
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a music library album.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.album.musicAlbum'
str - the DIDL Lite class for this object.

tag = 'container'
str - the XML element tag name used for this instance.

_translation = {'artist': ('upnp', 'artist'), 'creator': ('dc', 'creator'), 'long_description': ('upnp', 'longDescription'),
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a Sonos favorite music library album.

This class is not part of the DIDL spec and is Sonos specific.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.album.musicAlbum.sonos-favorite'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'contributor': ('dc', 'contributor'), 'long_description': ('upnp', 'longDesc
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a Sonos favorite music library compilation.

This class is not part of the DIDL spec and is Sonos specific.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.album.musicAlbum.compilation'
str - the DIDL Lite class for this object.

tag = 'container'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'contributor': ('dc', 'contributor'), 'long_description': ('upnp', 'longDesc
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

A content directory class representing a person.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.person'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'language': ('dc', 'language'), 'write_status': ('upnp', 'writeStatus')}
dfdf

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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a music library composer.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.

- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default 'RINCON_AssociatedZPUDN'. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the _translation class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.person.composer'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'language': ('dc', 'language'), 'write_status': ('upnp', 'writeStatus')}
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. Each key an attribute name and each key is a (namespace, tag) tuple.

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

Class that represents a music library artist.

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. Default True
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default 'RINCON_AssociatedZPUDN'. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the _translation class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.person.musicArtist'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'artist_discography_uri': ('upnp', 'artistDiscographyURI'), 'genre': ('upnp', 'genre'), 'creator': ('dc
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. Each key an attribute name and each key is a (namespace, tag) tuple.

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

Class that represents a music library album list.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.albumlist'
str - the DIDL Lite class for this object.

tag = 'container'
str - the XML element tag name used for this instance.

_translation = {'creator': ('dc', 'creator'), 'write_status': ('upnp', 'writeStatus')}
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a music library play list.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.playlistContainer'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'producer': ('dc', 'producer'), 'creator': ('dc', 'creator'), 'contributor': ('dc', 'contributor'), 'long_d
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

item_class = 'object.container.playlistContainer'

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

Class that represents all tracks by a single artist.

This type is returned by browsing an artist or a composer

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.playlistContainer.sameArtist'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'producer': ('dc', 'producer'), 'creator': ('dc', 'creator'), 'contributor': ('dc', 'contributor'), 'long_d
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

A content directory class representing a general genre.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.genre'
str - the DIDL Lite class for this object.

tag = 'container'
str - the XML element tag name used for this instance.

_translation = {'long_description': ('upnp', 'longDescription'), 'genre': ('upnp', 'genre'), 'creator': ('dc', 'creator'),
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

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

Class that represents a music genre.

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. Default `True`
- **resources** (*list, optional*) – a list of resources for this object.
- **None.** (*Default*) –
- **desc** (*str*) – A DIDL descriptor, default `'RINCON_AssociatedZPUDN'`. This is not the same as "description". It is used for identifying the relevant third party music service.
- ****kwargs** – Extra metadata. What is allowed depends on the `_translation` class attribute, which in turn depends on the DIDL class.

item_class = 'object.container.genre.musicGenre'
str - the DIDL Lite class for this object.

tag = 'item'
str - the XML element tag name used for this instance.

_translation = {'long_description': ('upnp', 'longDescription'), 'genre': ('upnp', 'genre'), 'creator': ('dc', 'creator'),
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. Each key an attribute name and each key is a `(namespace, tag)` tuple.

class soco.data_structures.**ListOfMusicInfoItems**(*items*, *number_returned*, *total_matches*, *update_id*)

Abstract container class for a list of music information items.

__getitem__(*key*)
Legacy get metadata by string key or list item(s) by index.

Deprecated since version 0.8: This overriding form of __getitem__ will be removed in the 3rd release after 0.8. The metadata can be fetched via the named attributes.

number_returned
str: the number of returned matches.

total_matches
str: the number of total matches.

update_id
str: the update ID.

class soco.data_structures.**SearchResult**(*items*, *search_type*, *number_returned*, *total_matches*, *update_id*)

Container class that represents a search or browse result.

Browse is just a special case of search.

search_type
str: the search type.

class soco.data_structures.**Queue**(*items*, *number_returned*, *total_matches*, *update_id*)
Container class that represents a queue.

soco.discovery module

This module contains methods for discovering Sonos devices on the network.

soco.discovery.**discover**(*timeout=5*, *include_invisible=False*, *interface_addr=None*)
Discover Sonos zones on the local network.

Return a 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,

Unexpected indentation.

after which return None if no zones found.

Parameters

- **timeout** (*int, optional*) – block for this many seconds, at most. Defaults to 5.
- **include_invisible** (*bool, optional*) – include invisible zones in the return set. Defaults to False.
- **interface_addr** (*str or None*) – 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 socket.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. Defaults to None.

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

---

soco.discovery.**any_soco**()
Return any visible soco device, for when it doesn't matter which.

Try to obtain an existing instance, or use *discover* if necessary. Note that this assumes that the existing instance has not left the network.

Returns A *SoCo* instance (or subclass if *config.SOCO_CLASS* is set, or None if no instances are found

Return type *SoCo*

soco.events module

Classes to handle Sonos UPnP Events and Subscriptions.

soco.events.**parse_event_xml**(*xml_event*)
Parse the body of a UPnP event.

Parameters **xml_event** (*bytes*) – bytes containing the body of the event encoded with utf-8.

Returns

A dict with keys representing the evented variables. The relevant value will usually be a string representation of the variable's value, but may on occasion be:

- a dict (eg when the volume changes, the value will itself be a dict containing the volume for each channel: `{'Volume': {'LF': '100', 'RF': '100', 'Master': '36'}}`)
- an instance of a *DidlObject* subclass (eg if it represents track metadata).

Return type dict

Example

Run this code, and change your volume, tracks etc:

```
from __future__ import print_function
try:
    from queue import Empty
except:  # Py2.7
    from Queue import Empty

import soco
from pprint import pprint
from soco.events import event_listener
# pick a device at random
device = soco.discover().pop()
print (device.player_name)
sub = device.renderingControl.subscribe()
sub2 = device.avTransport.subscribe()

while True:
    try:
        event = sub.events.get(timeout=0.5)
        pprint (event.variables)
    except Empty:
        pass
    try:
        event = sub2.events.get(timeout=0.5)
        pprint (event.variables)
    except Empty:
        pass

    except KeyboardInterrupt:
        sub.unsubscribe()
        sub2.unsubscribe()
        event_listener.stop()
        break
```

class soco.events.**Event**(*sid*, *seq*, *service*, *timestamp*, *variables=None*)
A read-only object representing a received event.

The values of the evented variables can be accessed via the `variables` dict, or as attributes on the instance itself. You should treat all attributes as read-only.

Parameters

- **sid** (*str*) – the subscription id.
- **seq** (*str*) – the event sequence number for that subscription.
- **timestamp** (*str*) – the time that the event was received (from Python's `time.time` function).
- **service** (*str*) – the service which is subscribed to the event.
- **variables** (*dict, optional*) – contains the `{names: values}` of the evented variables. Defaults to `None`.

Raises `AttributeError` – Not all attributes are returned with each event. An `AttributeError` will be raised if you attempt to access as an attribute a variable which was not returned in the event.

Example

```
>>> print event.variables['transport_state']
'STOPPED'
>>> print event.transport_state
'STOPPED'
```

__setattr__(*name*, *value*)
Disable (most) attempts to set attributes.

This is not completely foolproof. It just acts as a warning! See `object.__setattr__`.

class soco.events.**EventServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
A TCP server which handles each new request in a new thread.

Constructor. May be extended, do not override.

class soco.events.**EventNotifyHandler**(*request*, *client_address*, *server*)
Handles HTTP `NOTIFY` Verbs sent to the listener server.

do_NOTIFY()
Serve a `NOTIFY` request.

A `NOTIFY` request will be sent by a Sonos device when a state variable changes. See the UPnP Spec §4.3 [pdf] for details.

class soco.events.**EventServerThread**(*address*)
The thread in which the event listener server will run.

Parameters **address** (*tuple*) – The (ip, port) address on which the server should listen.

stop_flag = None
`threading.Event`: Used to signal that the server should stop.

address = None
`tuple`: The (ip, port) address on which the server is configured to listen.

run()
Start the server on the local IP at port 1400 (default).

Handling of requests is delegated to an instance of the *EventNotifyHandler* class.

class soco.events.**EventListener**
The Event Listener.

Runs an http server in a thread which is an endpoint for NOTIFY requests from Sonos devices.

is_running = None
bool: Indicates whether the server is currently running

address = None
tuple: The address (ip, port) on which the server is configured to listen.

start(*any_zone*)
Start the event listener listening on the local machine at port 1400 (default)

Make sure that your firewall allows connections to this port

Parameters **any_zone** (SoCo) – Any Sonos device on the network. It does not matter which device. It is used only to find a local IP address reachable by the Sonos net.

Note: The port on which the event listener listens is configurable. See *config.EVENT_LISTENER_PORT*

---

stop()
Stop the event listener.

class soco.events.**Subscription**(*service*, *event_queue=None*)
A class representing the subscription to a UPnP event.

Parameters

- **service** (Service) – The SoCo *Service* to which the subscription should be made.
- **event_queue** (Queue) – A queue on which received events will be put. If not specified, a queue will be created and used.

sid = None
str: A unique ID for this subscription

timeout = None
int: The amount of time in seconds until the subscription expires.

is_subscribed = None
bool: An indication of whether the subscription is subscribed.

events = None
Queue: The queue on which events are placed.

requested_timeout = None
int: The period (seconds) for which the subscription is requested

subscribe(*requested_timeout=None*, *auto_renew=False*)
Subscribe to the service.

If requested_timeout is provided, a subscription valid for that number of seconds will be requested, but not guaranteed. Check *timeout* on return to find out what period of validity is actually allocated.

Note: SoCo will try to unsubscribe any subscriptions which are still subscribed on program termination, but it is good practice for you to clean up by making sure that you call *unsubscribe()* yourself.

Parameters

- **requested_timeout** (*int, optional*) – The timeout to be requested.
- **auto_renew** (*bool, optional*) – If True, renew the subscription automatically shortly before timeout. Default False.

renew(*requested_timeout=None*)
Renew the event subscription.

You should not try to renew a subscription which has been unsubscribed, or once it has expired.

Parameters **requested_timeout** (*int, optional*) – The period for which a renewal request should be made. If None (the default), use the timeout requested on subscription.

unsubscribe()
Unsubscribe from the service's events.

Once unsubscribed, a Subscription instance should not be reused

time_left
int: The amount of time left until the subscription expires (seconds)

If the subscription is unsubscribed (or not yet subscribed), *time_left* is 0.

soco.exceptions module

Exceptions that are used by SoCo.

exception soco.exceptions.**SoCoException**
Base class for all SoCo exceptions.

exception soco.exceptions.**UnknownSoCoException**
An unknown UPnP error.

The exception object will contain the raw response sent back from the speaker as the first of its args.

exception soco.exceptions.**SoCoUPnPException**(*message*, *error_code*, *error_xml*, *error_description=''*)
A UPnP Fault Code, raised in response to actions sent over the network.

Parameters

- **message** (*str*) – The message from the server.
- **error_code** (*str*) – The UPnP Error Code as a string.
- **error_xml** (*str*) – The xml containing the error, as a utf-8 encoded string.
- **error_description** (*str*) – A description of the error. Default is ""

exception soco.exceptions.**CannotCreateDIDLMetadata**
Deprecated since version 0.11: Use *DIDLMetadataError* instead.

exception soco.exceptions.**DIDLMetadataError**
Raised if a data container class cannot create the DIDL metadata due to missing information.

For backward compatibility, this is currently a subclass of *CannotCreateDIDLMetadata*. In a future version, it will likely become a direct subclass of *SoCoException*.

exception soco.exceptions.**MusicServiceException**
An error relating to a third party music service.

exception soco.exceptions.**UnknownXMLStructure**
Raised if XML with an unknown or unexpected structure is returned.

exception soco.exceptions.**SoCoSlaveException**
Raised when a master command is called on a slave.

soco.groups module

This module contains classes and functionality relating to Sonos Groups.

class soco.groups.**ZoneGroup**(*uid*, *coordinator*, *members=None*)
A class representing a Sonos Group. It looks like this:

```
ZoneGroup(
    uid='RINCON_000FD584236D01400:58',
    coordinator=SoCo("192.168.1.101"),
    members=set([SoCo("192.168.1.101"), SoCo("192.168.1.102")])
)
```

Any SoCo instance can tell you what group it is in:

```
>>> device = soco.discovery.any_soco()
>>> device.group
ZoneGroup(
    uid='RINCON_000FD584236D01400:58',
    coordinator=SoCo("192.168.1.101"),
    members=set([SoCo("192.168.1.101"), SoCo("192.168.1.102")])
)
```

From there, you can find the coordinator for the current group:

```
>>> device.group.coordinator
SoCo("192.168.1.101")
```

or, for example, its name:

```
>>> device.group.coordinator.player_name
Kitchen
```

or a set of the members:

```
>>> device.group.members
{SoCo("192.168.1.101"), SoCo("192.168.1.102")}
```

For convenience, ZoneGroup is also a container:

```
>>> for player in device.group:
...   print player.player_name
Living Room
Kitchen
```

If you need it, you can get an iterator over all groups on the network:

```
>>> device.all_groups
<generator object all_groups at 0x108cf0c30>
```

A consistent readable label for the group members can be returned with the *label* and *short_label* properties.

Parameters

- **uid** (*str*) – The unique Sonos ID for this group, eg RINCON_000FD584236D01400:5.
- **coordinator** (SoCo) – The SoCo instance representing the coordinator of this group.
- **members** (*Iterable[SoCo]*) – An iterable containing SoCo instances which represent the members of this group.

uid = None
The unique Sonos ID for this group

coordinator = None
The *SoCo* instance which coordinates this group

members = None
A set of *SoCo* instances which are members of the group

label
str: A description of the group.

```
>>> device.group.label
'Kitchen, Living Room'
```

short_label
str: A short description of the group.

```
>>> device.group.short_label
'Kitchen + 1'
```

soco.ms_data_structures module

This module contains all the data structures for music service plugins.

soco.ms_data_structures.**get_ms_item**(*xml*, *service*, *parent_id*)
Return the music service item that corresponds to xml.

The class is identified by getting the type from the 'itemType' tag

soco.ms_data_structures.**tags_with_text**(*xml*, *tags=None*)
Return a list of tags that contain text retrieved recursively from an XML tree.

class soco.ms_data_structures.**MusicServiceItem**(***kwargs*)
Class that represents a music service item.

classmethod from_xml(*xml*, *service*, *parent_id*)
Return a Music Service item generated from xml.

Parameters

- **xml** (xml.etree.ElementTree.Element) – Object XML. All items containing text are added to the content of the item. The class variable valid_fields of each of the classes list the valid fields (after translating the camel case to underscore notation). Required fields are listed in the class variable by that name (where 'id' has been renamed to 'item_id').

- **service** (Instance of sub-class of *soco.plugins.SoCoPlugin*) – The music service (plugin) instance that retrieved the element. This service must contain `id_to_extended_id` and `form_uri` methods and `description` and `service_id` attributes.
- **parent_id** (*str*) – The parent ID of the item, will either be the extended ID of another MusicServiceItem or of a search

For a track the XML can e.g. be on the following form:

```
<mediaMetadata xmlns="http://www.sonos.com/Services/1.1">
  <id>trackid_141359</id>
  <itemType>track</itemType>
  <mimeType>audio/aac</mimeType>
  <title>Teacher</title>
  <trackMetadata>
    <artistId>artistid_10597</artistId>
    <artist>Jethro Tull</artist>
    <composerId>artistid_10597</composerId>
    <composer>Jethro Tull</composer>
    <albumId>albumid_141358</albumId>
    <album>MU - The Best Of Jethro Tull</album>
    <albumArtistId>artistid_10597</albumArtistId>
    <albumArtist>Jethro Tull</albumArtist>
    <duration>229</duration>
    <albumArtURI>http://varnish01.music.aspiro.com/sca/
     imscale?h=90&amp;w=90&amp;img=/content/music10/prod/wmg/
     1383757201/094639008452_20131105025504431/resources/094639008452.
     jpg</albumArtURI>
    <canPlay>true</canPlay>
    <canSkip>true</canSkip>
    <canAddToFavorites>true</canAddToFavorites>
  </trackMetadata>
</mediaMetadata>
```

classmethod from_dict(*dict_in*)

Initialize the class from a dict.

Parameters **dict_in** (*dict*) – The dictionary that contains the item content. Required fields are listed class variable by that name

__eq__(*playable_item*)

Return the equals comparison result to another `playable_item`.

__ne__(*playable_item*)

Return the not 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.

to_dict
Return a copy of the content dict.

didl_metadata
Return the DIDL metadata for a Music Service Track.

The metadata is on the form:

```
<DIDL-Lite xmlns:dc="http://purl.org/dc/elements/1.1/"
     xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/"
     xmlns:r="urn:schemas-rinconnetworks-com:metadata-1-0/"
     xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/">
  <item id="...self.extended_id..."
     parentID="...self.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/">
      self.content['description']
    </desc>
  </item>
</DIDL-Lite>
```

item_id
Return the item id.

extended_id
Return the extended id.

title
Return the title.

service_id
Return the service ID.

can_play
Return a boolean for whether the item can be played.

parent_id
Return the extended parent_id, if set, otherwise return None.

album_art_uri
Return the album art URI if set, otherwise return None.

class soco.ms_data_structures.**MSTrack**(*title*, *item_id*, *extended_id*, *uri*, *description*, *service_id*, ***kwargs*)
Class that represents a music service track.

Initialize MSTrack item.

album
Return the album title if set, otherwise return None.

artist
Return the artist if set, otherwise return None.

duration
Return the duration if set, otherwise return None.

uri
Return the URI.

class soco.ms_data_structures.**MSAlbum**(*title*, *item_id*, *extended_id*, *uri*, *description*, *service_id*, ***kwargs*)
Class that represents a Music Service Album.

artist
Return the artist if set, otherwise return None.

uri
Return the URI.

class soco.ms_data_structures.**MSAlbumList**(*title*, *item_id*, *extended_id*, *uri*, *description*, *service_id*, ***kwargs*)
Class that represents a Music Service Album List.

uri
Return the URI.

class soco.ms_data_structures.**MSPlaylist**(*title*, *item_id*, *extended_id*, *uri*, *description*, *service_id*, ***kwargs*)
Class that represents a Music Service Play List.

uri
Return the URI.

class soco.ms_data_structures.**MSArtistTracklist**(*title*, *item_id*, *extended_id*, *uri*, *description*, *service_id*, ***kwargs*)
Class that represents a Music Service Artist Track List.

uri
Return the URI.

class soco.ms_data_structures.**MSArtist**(*title*, *item_id*, *extended_id*, *service_id*, ***kwargs*)
Class that represents a Music Service Artist.

class soco.ms_data_structures.**MSFavorites**(*title*, *item_id*, *extended_id*, *service_id*, ***kwargs*)
Class that represents a Music Service Favorite.

class soco.ms_data_structures.**MSCollection**(*title*, *item_id*, *extended_id*, *service_id*, ***kwargs*)
Class that represents a Music Service Collection.

soco.music_library module

Access to the Music Library.

The Music Library is the collection of music stored on your local network. For access to third party music streaming services, see the *music_service* module.

class soco.music_library.**MusicLibrary**(*soco=None*)
The Music Library.

Parameters **soco** (*SoCo*, optional) – A *SoCo* instance to query for music library information. If None, or not supplied, a random *SoCo* instance will be used.

get_artists(**args*, ***kwargs*)
Convenience method for *get_music_library_information* with search_type='artists'. For details of other arguments, see *that method*.

get_album_artists(**args*, ***kwargs*)
Convenience method for *get_music_library_information* with `search_type='album_artists'`. For details of other arguments, see *that method*.

get_albums(**args*, ***kwargs*)
Convenience method for *get_music_library_information* with `search_type='albums'`. For details of other arguments, see *that method*.

get_genres(**args*, ***kwargs*)
Convenience method for *get_music_library_information* with `search_type='genres'`. For details of other arguments, see *that method*.

get_composers(**args*, ***kwargs*)
Convenience method for *get_music_library_information* with `search_type='composers'`. For details of other arguments, see *that method*.

get_tracks(**args*, ***kwargs*)
Convenience method for *get_music_library_information* with `search_type='tracks'`. For details of other arguments, see *that method*.

get_playlists(**args*, ***kwargs*)
Convenience method for *get_music_library_information* with `search_type='playlists'`. For details of other arguments, see *that method*.

Note: The playlists that are referred to here are the playlists imported from the music library, they are not the Sonos playlists.

---

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 have to do with paging of the results. By 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** (*str*) – The kind of information to retrieve. Can be one of: 'artists', 'album_artists', 'albums', 'genres', 'composers', 'tracks', 'share', 'sonos_playlists', or 'playlists', where playlists are the imported playlists from the music library.
- **start** (*int, optional*) – starting number of returned matches (zero based). Default 0.
- **max_items** (*int, optional*) – Maximum number of returned matches. Default 100.
- **full_album_art_uri** (*bool*) – whether the album art URI should be absolute (i.e. including the IP address). Default False.
- **search_term** (*str, optional*) – 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** (*str, optional*) – A list of strings that indicate one or more subcategories to dive into.
- **complete_result** (*bool*) – if True, 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 an instance of *SearchResult*.

Return type *SearchResult*

Note:

•The maximum numer of results may be restricted by the unit, presumably due to transfer size consideration, so check the returned number against that requested.

•The playlists that are returned with the 'playlists' search, are the playlists imported from the music library, they are not the Sonos playlists.

Raises *SoCoException* upon errors.

browse(*ml_item=None*, *start=0*, *max_items=100*, *full_album_art_uri=False*, *search_term=None*, *subcategories=None*)

Browse (get sub-elements from) a music library item.

Parameters

- **ml_item** (*DidlItem*) – the item to browse, if left out or None, items at the root level will be searched.
- **start** (*int*) – the starting index of the results.

- **max_items** (*int*) – the maximum number of items to return.
- **full_album_art_uri** (*bool*) – whether the album art URI should be fully qualified with the relevant 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 descend into. Note: Providing sub categories will not work if `ml_item` is `None`.

Returns A *SearchResult* instance.

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 from) a given music library item, specified by a string.

Parameters

- **search_type** (*str*) – 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** (*str*) – a term to search for.
- **start** (*int*) – starting number of returned matches. Default 0.
- **max_items** (*int*) – Maximum number of returned matches. Default 100.
- **full_album_art_uri** (*bool*) – whether the album art URI should be absolute (i.e. including the IP address). Default `False`.

Returns a *SearchResult* instance.

Return type *SearchResult*

Note: The maximum numer of results may be restricted by the unit, presumably due to transfer size consideration, so check the returned number against that requested.

---

library_updating
bool: whether the music library is in the process of being updated.

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

Parameters **album_artist_display_option** (*str*) – a value for the album artist compilation setting (see *album_artist_display_option*).

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

Parameters

- **artist** (*str*) – an artist's name.
- **album** (*str, optional*) – an album name. Default `None`.

- **track** (*str, optional*) – a track name. Default None.
- **full_album_art_uri** (*bool*) – whether the album art URI should be absolute (i.e. including the IP address). Default False.

Returns A *SearchResult* instance.

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

Parameters

- **artist** (*str*) – an artist's name.
- **full_album_art_uri** – whether the album art URI should be absolute (i.e. including the IP address). Default False.

Returns A *SearchResult* instance.

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

Parameters

- **artist** (*str*) – an artist's name.
- **album** (*str*) – an album name.
- **full_album_art_uri** – whether the album art URI should be absolute (i.e. including the IP address). Default False.

Returns A *SearchResult* instance.

album_artist_display_option
str: The current value of the album artist compilation setting.

Possible values are:

- 'WMP' - use Album Artists
- 'ITUNES' - use iTunes® Compilations
- 'NONE' - do not group compilations

See also:

The Sonos FAQ on compilation albums.

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

soco.services module

Classes representing Sonos UPnP services.

```
>>> import soco
>>> device = soco.SoCo('192.168.1.102')
>>> print(RenderingControl(device).GetMute([('InstanceID', 0),
...     ('Channel', 'Master')]))
{'CurrentMute': '0'}
>>> r = ContentDirectory(device).Browse([
...    ('ObjectID', 'Q:0'),
...    ('BrowseFlag', 'BrowseDirectChildren'),
...    ('Filter', '*'),
...    ('StartingIndex', '0'),
...    ('RequestedCount', '100'),
```

```
...     ('SortCriteria', '')
...     ])
>>> print(r['Result'])
<?xml version="1.0" ?><DIDL-Lite xmlns="urn:schemas-upnp-org:metadata ...
>>> for action, in_args, out_args in AlarmClock(device).iter_actions():
...     print(action, in_args, out_args)
...
SetFormat [Argument(name='DesiredTimeFormat', vartype='string'), Argument(
name='DesiredDateFormat', vartype='string')] []
GetFormat [] [Argument(name='CurrentTimeFormat', vartype='string'),
Argument(name='CurrentDateFormat', vartype='string')] ...
```

class soco.services.**Action**(*name*, *in_args*, *out_args*)
A UPnP Action and its arguments.

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *name*, *in_args*, *out_args*)
Create new instance of Action(name, in_args, out_args)

__repr__()
Return a nicely formatted representation string

in_args
Alias for field number 1

name
Alias for field number 0

out_args
Alias for field number 2

class soco.services.**Argument**(*name*, *vartype*)
A UPnP Argument and its type.

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *name*, *vartype*)
Create new instance of Argument(name, vartype)

__repr__()
Return a nicely formatted representation string

name
Alias for field number 0

vartype
Alias for field number 1

class soco.services.**Service**(*soco*)
A class representing a UPnP service.

This is the base class for all Sonos Service classes. This class has a dynamic method dispatcher. Calls to methods which are not explicitly defined here are dispatched automatically to the service action with the same name.

Parameters **soco** (SoCo) – A *SoCo* instance to which the UPnP Actions will be sent

soco = None
SoCo: The *SoCo* instance to which UPnP Actions are sent

service_type = None
str: The UPnP service type.

version = None
str: The UPnP service version.

base_url = None
str: The base URL for sending UPnP Actions.

control_url = None
str: The UPnP Control URL.

scpd_url = None
str: The service control protocol description URL.

event_subscription_url = None
str: The service eventing subscription URL.

cache = None
A cache for storing the result of network calls. By default, this is a *TimedCache* with a default timeout=0.

__getattr__(*action*)
Called when a method on the instance cannot be found.

Causes an action to be sent to UPnP server. See also `object.__getattr__`.

Parameters **action** (*str*) – The name of the unknown method.

Returns The callable to be invoked. .

Return type callable

static **wrap_arguments**(*args=None*)
Wrap a list of tuples in xml ready to pass into a SOAP request.

Parameters **args** (*list*) – a list of (name, value) tuples specifying the name of each argument and its value, eg `[('InstanceID', 0), ('Speed', 1)]`. The value can be a string or something with a string representation. The arguments are escaped and wrapped in <name> and <value> tags.

Example

```
>>> from soco import SoCo
>>> device = SoCo('192.168.1.101')
>>> s = Service(device)
>>> print(s.wrap_arguments([('InstanceID', 0), ('Speed', 1)]))
<InstanceID>0</InstanceID><Speed>1</Speed>'
```

static **unwrap_arguments**(*xml_response*)
Extract arguments and their values from a SOAP response.

Parameters **xml_response** (*str*) – SOAP/xml response text (unicode, not utf-8).

Returns a dict of `{argument_name, value)}` items.

Return type dict

build_command(*action*, *args=None*)
Build a SOAP request.

Parameters

- **action** (*str*) – the name of an action (a string as specified in the service description XML file) to be sent.
- **args** (*list, optional*) – Relevant arguments as a list of (name, value) tuples.

Returns a tuple containing the POST headers (as a dict) and a string containing the relevant SOAP body. Does not set content-length, or host headers, which are completed upon sending.

Return type tuple

send_command(*action*, *args=None*, *cache=None*, *cache_timeout=None*)
Send a command to a Sonos device.

Parameters

- **action** (*str*) – the name of an action (a string as specified in the service description XML file) to be sent.
- **args** (*list, optional*) – Relevant arguments as a list of (name, value) tuples.
- **cache** (Cache) – A cache is operated so that the result will be stored for up to `cache_timeout` seconds, and a subsequent call with the same arguments within that period will be returned from the cache, saving a further network call. The cache may be invalidated or even primed from another thread (for example if a UPnP event is received to indicate that the state of the Sonos device has changed). If `cache_timeout` is missing or `None`, the cache will use a default value (which may be 0 - see *cache*). By default, the cache identified by the service's *cache* attribute will be used, but a different cache object may be specified in the *cache* parameter.

Returns a dict of `{argument_name, value)}` items.

Return type dict

Raises

- *SoCoUPnPException* – if a SOAP error occurs.
- *UnknownSoCoException* – if an unknonwn UPnP error occurs.
- `requests.exceptions.HTTPError` – if an http error.

handle_upnp_error(*xml_error*)
Disect a UPnP error, and raise an appropriate exception.

Parameters **xml_error** (*str*) – a unicode string containing the body of the UPnP/SOAP Fault response. Raises an exception containing the error code.

subscribe(*requested_timeout=None*, *auto_renew=False*, *event_queue=None*)
Subscribe to the service's events.

Parameters

- **requested_timeout** (*int, optional*) – If requested_timeout is provided, a subscription valid for that number of seconds will be requested, but not guaranteed. Check *Subscription.timeout* on return to find out what period of validity is actually allocated.
- **auto_renew** (*bool*) – If auto_renew is `True`, the subscription will automatically be renewed just before it expires, if possible. Default is `False`.

- **event_queue** (Queue) – a thread-safe queue object on which received events will be put. If not specified, a (Queue) will be created and used.

Returns an insance of *Subscription*, representing the new subscription.

Return type *Subscription*

To unsubscribe, call the *unsubscribe* method on the returned object.

iter_actions()
Yield the service's actions with their arguments.

Yields *Action* – the next action.

Each action is an Action namedtuple, consisting of action_name (a string), in_args (a list of Argument namedtuples consisting of name and argtype), and out_args (ditto), eg:

```
Action(
    name='SetFormat',
    in_args=[
        Argument(name='DesiredTimeFormat', vartype='string'),
        Argument(name='DesiredDateFormat', vartype='string')],
    out_args=[]
)
```

iter_event_vars()
Yield the services eventable variables.

Yields tuple – a tuple of (variable name, data type).

class soco.services.**AlarmClock**(*soco*)
Sonos alarm service, for setting and getting time and alarms.

class soco.services.**MusicServices**(*soco*)
Sonos music services service, for functions related to 3rd party music services.

class soco.services.**DeviceProperties**(*soco*)
Sonos device properties service, for functions relating to zones, LED state, stereo pairs etc.

class soco.services.**SystemProperties**(*soco*)
Sonos system properties service, for functions relating to authentication etc.

class soco.services.**ZoneGroupTopology**(*soco*)
Sonos zone group topology service, for functions relating to network topology, diagnostics and updates.

GetZoneGroupState(**args*, ***kwargs*)
Overrides default handling to use the global shared zone group state cache, unless another cache is specified.

class soco.services.**GroupManagement**(*soco*)
Sonos group management service, for services relating to groups.

class soco.services.**QPlay**(*soco*)
Sonos Tencent QPlay service (a Chinese music service)

class soco.services.**ContentDirectory**(*soco*)
UPnP standard Content Directory service, for functions relating to browsing, searching and listing available music.

class soco.services.**MS_ConnectionManager**(*soco*)
UPnP standard connection manager service for the media server.

class soco.services.**RenderingControl**(*soco*)
UPnP standard rendering control service, for functions relating to playback rendering, eg bass, treble, volume and EQ.

class soco.services.**MR_ConnectionManager**(*soco*)
UPnP standard connection manager service for the media renderer.

class soco.services.**AVTransport**(*soco*)
UPnP standard AV Transport service, for functions relating to transport management, eg play, stop, seek, playlists etc.

class soco.services.**Queue**(*soco*)
Sonos queue service, for functions relating to queue management, saving queues etc.

class soco.services.**GroupRenderingControl**(*soco*)
Sonos group rendering control service, for functions relating to group volume etc.

soco.snapshot module

Functionality to support saving and restoring the current Sonos state.

This is useful for scenarios such as when you want to switch to radio and then back again to what was playing previously.

class soco.snapshot.**Snapshot**(*device*, *snapshot_queue=False*)
A snapshot of the current state.

Note: This does not change anything to do with the configuration such as which group the speaker is in, just settings that impact what is playing, or how it is played.

List of sources that may be playing using root of media_uri:

`x-rincon-queue`: playing from Queue
`x-sonosapi-stream`: playing a stream (eg radio)
`x-file-cifs`: playing file
`x-rincon`: slave zone (only change volume etc. rest from coordinator)

Parameters

- **device** (SoCo) – The device to snapshot
- **snapshot_queue** (*bool*) – Whether the queue should be snapshotted. Defaults to `False`.

Warning: It is strongly advised that you do not snapshot the queue unless you really need to as it takes a very long time to restore large queues as it is done one track at a time.

snapshot()
Record and store the current state of a device.

Returns `True` if the device is a coordinator, `False` otherwise. Useful for determining whether playing an alert on a device will ungroup it.

Return type bool

restore(*fade=False*)
Restore the state of a device to that which was previously saved.

For coordinator devices restore everything. For slave devices only restore volume etc., not transport info (transport info comes from the slave's coordinator).

Parameters **fade** (*bool*) – Whether volume should be faded up on restore.

soco.soap module

Classes for handling SoCo's basic SOAP requirements.

This module does not handle anything like the full SOAP Specification , but is enough for SoCo's needs. Sonos uses SOAP for UPnP communications, and for communication with third party music services.

exception soco.soap.**SoapFault**(*faultcode*, *faultstring*, *detail=None*)
An exception encapsulating a SOAP Fault.

Parameters

- **faultcode** (*str*) – The SOAP faultcode.
- **faultstring** (*str*) – The SOAP faultstring.
- **detail** (Element) – The SOAP fault detail, as an ElementTree Element. Defaults to None.

class soco.soap.**SoapMessage**(*endpoint*, *method*, *parameters=None*, *http_headers=None*, *soap_action=None*, *soap_header=None*, *namespace=None*, ***request_args*)
A SOAP Message representing a remote procedure call.

Uses the Requests library for communication with a SOAP server.

Parameters

- **endpoint** (*str*) – The SOAP endpoint URL for this client.
- **method** (*str*) – The name of the method to call.
- **parameters** (*list*) – A list of (name, value) tuples containing the parameters to pass to the method. Default None.
- **http_headers** (*dict*) – A dict in the form {'Header': 'Value,..} containing http headers to use for the http request. Content-type and SOAPACTION headers will be created automatically, so do not include them here. Use this, for example, to set a user-agent.
- **soap_action** (*str*) – The value of the SOAPACTION header. Default 'None'.
- **soap_header** (*str*) – A string representation of the XML to be used for the SOAP Header. Default None.
- **namespace** (*str*) – The namespace URI to use for the method and parameters. None, by default.
- ****request_args** – Other keyword parameters will be passed to the Requests request which is used to handle the http communication. For example, a timeout value can be set.

prepare_headers(*http_headers*, *soap_action*)
Prepare the http headers for sending.

Add the SOAPACTION header to the others.

Parameters

- **http_headers** (*dict*) – A dict in the form {'Header': 'Value,..} containing http headers to use for the http request.
- **soap_action** (*str*) – The value of the SOAPACTION header.

Returns headers including the SOAPACTION header.

Return type dict

prepare_soap_header(*soap_header*)
Prepare the SOAP header for sending.

Wraps the soap header in appropriate tags.

Parameters **soap_header** (*str*) – A string representation of the XML to be used for the SOAP Header

Returns The soap header wrapped in appropriate tags.

Return type str

prepare_soap_body(*method*, *parameters*, *namespace*)
Prepare the SOAP message body for sending.

Parameters

- **method** (*str*) – The name of the method to call.
- **parameters** (*list*) – A list of (name, value) tuples containing the parameters to pass to the method.
- **namespace** (*str*) – tThe XML namespace to use for the method.

Returns A properly formatted SOAP Body.

Return type str

prepare_soap_envelope(*prepared_soap_header*, *prepared_soap_body*)
Prepare the SOAP Envelope for sending.

Parameters

- **prepared_soap_header** (*str*) – A SOAP Header prepared by *prepare_soap_header*
- **prepared_soap_body** (*str*) – A SOAP Body prepared by *prepare_soap_body*

Returns A prepared SOAP Envelope

Return type str

prepare()
Prepare the SOAP message for sending to the server.

call()
Call the SOAP method on the server.

Returns the decapusulated SOAP response from the server, still encoded as utf-8.

Return type str

Raises

- *SoapFault* – if a SOAP error occurs.
- HTTPError – if an http error occurs.

soco.utils module

This class contains utility functions used internally by SoCo.

soco.utils.**really_unicode**(*in_string*)
Make a string unicode. Really.

Ensure `in_string` is returned as unicode through a series of progressively relaxed decodings.

Parameters **in_string** (*str*) – The string to convert.

Returns Unicode.

Return type str

Raises `ValueError`

soco.utils.**really_utf8**(*in_string*)
Encode a string with utf-8. Really.

First decode `in_string` via *really_unicode* to ensure it can successfully be encoded as utf-8. This is required since just calling encode on a string will often cause Python 2 to perform a coerced strict auto-decode as ascii first and will result in a `UnicodeDecodeError` being raised. After *really_unicode* returns a safe unicode string, encode as utf-8 and return the utf-8 encoded string.

Parameters **in_string** – The string to convert.

soco.utils.**camel_to_underscore**(*string*)
Convert camelcase to lowercase and underscore.

Recipe from http://stackoverflow.com/a/1176023

Parameters **string** (*str*) – The string to convert.

Returns The converted string.

Return type str

soco.utils.**prettify**(*unicode_text*)
Return a pretty-printed version of a unicode XML string.

Useful for debugging.

Parameters **unicode_text** (*str*) – A text representation of XML (unicode, *not* utf-8).

Returns A pretty-printed version of the input.

Return type str

soco.utils.**show_xml**(*xml*)
Pretty print an `ElementTree` XML object.

Parameters **xml** (`ElementTree`) – The `ElementTree` to pretty print

Note: This is used a convenience function used during development. It is not used anywhere in the main code base.

---

class soco.utils.**deprecated**(*since*, *alternative=None*, *will_be_removed_in=None*)
A decorator for marking deprecated objects.

Used internally by SoCo to cause a warning to be issued when the object is used, and marks the object as deprecated in the Sphinx documentation.

Parameters

- **since** (*str*) – The version in which the object is deprecated.
- **alternative** (*str, optional*) – The name of an alternative object to use
- **will_be_removed_in** (*str, optional*) – The version in which the object is likely to be removed.

Example

```
@deprecated(since="0.7", alternative="new_function")
def old_function(args):
    pass
```

soco.utils.**url_escape_path**(*path*)
Escape a string value for a URL request path.

Parameters **str** – The path to escape

Returns The escaped path

Return type str

```
>>> url_escape_path("Foo, bar & baz / the hackers")
u'Foo%2C%20bar%20%26%20baz%20%2F%20the%20hackers'
```

soco.xml module

This class contains XML related utility functions.

soco.xml.**NAMESPACES = {'upnp': 'urn:schemas-upnp-org:metadata-1-0/upnp/', '': 'urn:schemas-upnp-org:metadata-1-0/D**
Commonly used namespaces, and abbreviations, used by *ns_tag*.

soco.xml.**ns_tag**(*ns_id*, *tag*)
Return a namespace/tag item.

Parameters

- **ns_id** (*str*) – A namespace id, eg "dc" (see *NAMESPACES*)
- **tag** (*str*) – An XML tag, eg "author"

Returns A fully qualified tag.

Return type str

The ns_id is translated to a full name space via the *NAMESPACES* constant:

```
>>> xml.ns_tag('dc','author')
'{http://purl.org/dc/elements/1.1/}author'
```

1.10 SoCo releases

1.10.1 SoCo 0.12 release notes

SoCo 0.12 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

- New `MusicService` class for access to all the music services known to Sonos. Note that some of this code is still unstable, and in particular the data structures returned by methods such as `get_metadata` may change in future. (#262, #358)
- Add information to the docs about how to put SoCo in the Python path, for test execution (#327, #319)
- added `to_dict()` / `from_dict()` to `DidlResource` (#330, #318)
- All tests have been moved from the `unittests` directory to the tests directory (#336)
- For developers, more `make` targets, and a better sdist build (#346)
- Added `discovery.any_soco()`, for when you need a `SoCo` instance, but you don't care which. This is slightly better than the traditional `device = soco.discover().pop()` since it will return an existing instance if one is available, without sending discovery packets (#262)
- Modified `DidlObject.to_dict()` so that any associated resources list will be also returned as a list of dictionaries instead of returning a list of `DidlResource` objects. (#340, #338)
- Added a `sonosdump` tool in `dev_tools`, which can print out the various UPnP methods which Sonos uses (#344)
- Added methods for sonos playlist management: `reorder_sonos_playlist`, `clear_sonos_playlist`, `move_in_sonos_playlist`, `remove_from_sonos_playlist`, `get_sonos_playlist_by_attr` (#352, #348, #353) and `remove_sonos_playlist` (#341, #345)
- Support playmodes repeat-one (`REPEAT_ONE`) and shuffle-repeat-one (`SHUFFLE_REPEAT_ONE`) introduced by Sonos 6.0 (#387)
- Better discovery: SoCo tries harder to find devices on the local network, particularly where there are multiple network interfaces. The default discovery timeout is also increased to 5 seconds (#395, #432)
- Large work package on the docs, which contains a new front page, more sections, some advanced topics and an example page (#406, #360, #368, #362, #326, #369).
- Added optional timeout argument to be passed onto requests when getting speaker info (#302)
- Ignore `.#` specified subclasses in Didl xml. Several music services seem to use an out-of-spec way to make sub classes in Didl, by specifying the subclass name or function after a #. This caused our implementation of Didl to reject it. This has now been fixed by simple ignoring these un-official subclasses (#425)
- Added methods to manipulate sonos sleep functionality: `set_sleep_timer`, `get_sleep_timer` (#413)
- Various cleanups (#351)
- Extended `get_speaker_info` to return more information about the Sonos speakers (#335, #320)

Bugfixes

- Clear zone group cache and reparse zone group information after join and unjoin to prevent giving wrong topology information. (#323, #321)
- Fix typo preventing SoCo from parsing the audio metadata object used when a TV is playing. (#331)
- Fix bug where SoCo would raise an exception if music services sent metadata with invalid XML characters (#392, #386)
- Event lister was (incorrectly) responding to `GET` and `HEAD` requests, which could result in local files being served (#430)
- Minor fix because ordereddict.values in py3 return ValuesView (#359)

- Fixed bugs with parsing events (#276)
- Fixed unit tests (#343, #342)
- Fix in MusicLibrary constructor (#370)

Backwards Compatability

- Dropped support for Python 3.2 (#324)
- Methods relating to the music library (`get_artists`, `get_album_artists`, `get_albums` and others) have been moved to the `music_library` module. Instead of `device.get_album_artists()`, please now use `device.music_library.get_album_artists()` etc. Old code will continue to work for the moment, but will raise deprecation warnings (#350)
- Made a hard deprecation of the Spotify plugin since the API it relied on has been deprecated and it therefore no longer worked (#401, #423)
- Dropped pylint checks for Python 2.6 (#363)

1.10.2 SoCo 0.11.1 release notes

SoCo 0.11.1 is a new version of the SoCo library. This release fixes a bug with the installation of SoCo.

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

Bugfixes

- Installation fails on systems where the default encoding is not UTF-8 (#312, #313)

1.10.3 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)

1.10.4 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.

1.10.5 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.

1.10.6 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)

1.10.7 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 Music Library Data Structures* 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")
```

1.10.8 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.

1.11 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 properly 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.

1.11.1 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
```

1.11.2 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

Note: To run the unittests in this way, the soco package must be importable, i.e. the folder that contains it (the root folder of the git archive) must be in the list of paths that Python can import from (the PYTHONPATH). The easiest way to set this up, if you are using a virtual environment, is to install SoCo from the git archive in editable mode. This is done by executing the following command from the git archive root:

```
pip install -e .
```

1.11.3 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
```

1.11.4 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`

1.11.5 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.

1.11.6 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.

1.12 Release Procedures

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

1.12.1 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`.

1.12.2 Create and Publish

- Verify that all tests pass.
- Update the version number in `__init__.py` (see example).
- 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.

1.12.3 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

a

c

d

e

g

m

p

s

u

x

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

T

U

V

W

Z