soco Documentation

Release 0.21.2

Author

Feb 20, 2021

User Documentation

SoCo (Sonos Controller) is a high level Python 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 pique your interest: *Speaker Topologies*, *Events* and *UPnP Services*. Finally dive into the *the full module reference documentation*.

If you have a question, start by consulting the *FAQ*. If your question remains unanswered, post a question in the SoCo/SoCo Gitter chat room or in the SoCo Google group.

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*.

1.1.1.1 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.

1.1.1.2 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=73187104385f04d18ce3e56853be1e0c
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
```

1.1.1.3 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.

1.1.2.1 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.

1.1.2.2 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

1.2.1.1 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")
```

1.2.1.2 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")
```

1.2.1.3 Getting a named device

Getting a device by player name can be done with the *soco.discovery.by_name()* function:

```
>>> from soco.discovery import by_name
>>> device = by_name("Living Room")
>>> device
SoCo("192.168.1.18")
```

1.2.2 Playback control

1.2.2.1 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'
```

1.2.2.2 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'
```

1.2.3 Seeing and manipulating the queue

1.2.3.1 Getting the queue

Getting the queue is done with the *get_queue()* method:

```
>>> queue = device.get_queue()
>>> queue
Queue(items=[<DidlMusicTrack 'b'Blackened'' at 0x7f2237006dd8>, ..., <DidlMusicTrack
↪'b'Dyers Eve'' at 0x7f2237006828>])
```

The returned *Queue* object is a sequence of items from the queue, meaning that it can be iterated over and its length aquired with `len()`:

```
>>> len(queue)
9
>>> for item in queue:
...     print(item.title)
...
Blackened
...and Justice for All
Eye of the Beholder
One
The Shortest Straw
Harvester of Sorrow
The Frayed Ends of Sanity
To Live Is to Die
Dyers Eve
```

The queue object also has *total_matches* and *number_returned* attributes, which are used to figure out whether paging is required in order to get all elements of the queue. See the *ListOfMusicInfoItems* docstring for details.

1.2.3.2 Clearing the queue

Clearing the queue is done with the *clear_queue()* method as follows:

```
>>> queue = device.get_queue()
>>> len(queue)
9
>>> device.clear_queue()
>>> queue = device.get_queue()
>>> len(queue)
0
```

1.2.4 Listing and deleting music library shares

Music library shares are the local network drive shares connected to Sonos, which host the audio content in the Sonos Music Library.

To list the shares connected to Sonos, use the *list_library_shares()* method as follows:

```
>>> device.music_library.list_library_shares()
['//share_host_01/music', '//share_host_02/music']
```

The result is a list of network share locations.

To delete a network share, use the *delete_library_share()* method as follows:

```
>>> device.music_library.delete_library_share('//share_host_01/music')
```

You may want to check that the deletion has succeeded, by waiting a few seconds, then confirming that the share has disappeared from the list of shares.

1.3 Frequently Asked Questions

This page contains answers to a few commonly asked questions.

1.3.1 Why can't I play a URI from music service X with the `play_uri()` method?

The *play_uri()* method is only for playing URI's with un-restricted access such as podcasts, certain radion stations or sound clips on webpages. In short, the *play_uri()* method is for anything that will play as a sound file in your browser without authentication.

To play music from a music service, you will need to go via the *music_service* module. Here you can search or browse to obtain music service items, which can be added to the queue and played.

1.3.2 Why can't I add a URI from music service X to the queue with the `add_uri_to_queue()` method?

See *Why can't I play a URI from music service X with the play_uri() method?*.

1.3.3 Can I make my Sonos® speaker play music from my local hard drive with SoCo?

At the face of it, *no*. Sonos® devices can only play music that is available on the network i.e. can be reached via a URI. So you have two options:

1. You can share your local music folder onto the network and add it to the Sonos® library as a part of your music collection, which can then be searched, browsed and played with SoCo.
2. You can cheat and make Python serve the files on the fly and play them as URIs. The play local files example shows one way in which this can be accomplished.

> **Warning:** Note that this example is meant as a convenient way get started, but that no security precautions has been taken to e.g. prevent serving other files out into the local network. Take appropriate actions if this is a concern.

1.3.4 How can I save, then restore the previous playing Sonos state ?

This is useful for scenarios such as when you want to switch to radio, an announcement or doorbell sound and then back to what was playing previously. Documentation of the Snapshot *snapshot* module.

SoCo provides a snapshot module that captures the current state of a player and then when requested re-instates that state. Examples of it's use are:

- basic snap example
- multi zone example

1.4 Plugins

Plugins can extend the functionality of SoCo.

1.4.1 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.

1.4.2 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()
```

1.4.3 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.

1.5 Authors

1.5.1 Project Creator

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

1.5.2 Maintainers

- Lawrence Akka
- Stefan Kögl
- Kenneth Nielsen
- David Harding

1.5.3 Contributors

(alphabetical)

- Petter Aas
- Murali Allada
- Joel Björkman
- Aaron Daubman
- Johan Elmerfjord
- David Harding
- Jeff Hinrichs
- Jeroen Idserda

- Todd Neal
- nixscripter
- Kenneth Nielsen
- Dave O'Connor
- Dennnis O'Reilly
- phut
- Dan Poirier
- Jason Ting
- Peter Toft (pwt)
- Scott G Waters

1.6 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={SoCo("192.168.1.101"), SoCo("192.168.1.102")}
)
```

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

1.6.1 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>
```

1.7 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.

1.7.1 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:

1.7.2 Events

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

1.8 Events

1.8.1 The events_twisted module

The *soco.events_twisted* module has been provided for those wanting to use soco in an application built on the twisted framework who want the event listener also to be implemented using twisted. The *soco.events_twisted* page contains an example of how to use the module.

The event listener is an HTTP server that receives event notifications from sonos devices. In the *soco.events* module, it is implemented using threading and requests. The *soco.events* module will apply by default, unless *config.EVENTS_MODULE* is set to point to the *soco.events_twisted* module.

Twisted is not a soco dependency. The existence of the events_twisted module is not a recommendation or endorsement of twisted. The events_twisted module has been provided because there are some soco users who use twisted.

If you wish to use events_twisted, it is assumed you already use and are familiar with the twisted framework. No guidance is provided here on how to install or use twisted.

The main differences between *soco.events_twisted* and *soco.events* are:

- *soco.events_twisted* uses twisted, rather than requests, for making and receiving HTTP calls. Network calls in events_twisted return at once without blocking
- in *soco.events_twisted*, the event listener runs in the main thread of execution. Threading is not used
- *soco.events_twisted* requires a twisted reactor to be running in the application into which it is imported. It will not install or start a reactor
- *soco.events_twisted* is not threadsafe and should run in the main thread of execution. Therefore, subscribing to events should happen in the main thread of execution. In part, this is because a Deferred is not threadsafe
- in *soco.events_twisted*, if the requested port is not available, the event_listener will automatically try the next port, within a maximum range of 100 of the port initially requested
- in *soco.events_twisted*, subscribe, renew and unsubscribe return a Deferred the result of which will be the *soco.events_twisted.Subscription* instance. The Subscription can be accessed by adding a callback to receive it. In addition, Deferred.subscription is set to refer to the Subscription. This is a simpler and quicker way to get the Subscription
- in *soco.events_twisted*, Subscription.callback can be set to refer to a function that will be called each time a *soco.events_base.Event* is received by the Subscription. The callback will be passed the Event as the only parameter. This is likely to be the most convenient way to receive Events. If Subscription.callback is not set, or is not callable, Events will be put on the Subscription's event queue, in the same way as for the events module.

Please note that all network calls in soco (other than those in events_twisted) are made using the requests library, which blocks. In an application based on twisted, it may be desirable to make these network calls asynchronously, so they do not block. Two solutions to consider are (a) to use threads when calling other potentially blocking soco methods or (b) to use a subprocess to handle calls to soco. Twisted provides the deferToThread method for deferring potentially blocking methods to a thread. If a subprocess is to be used, there will need to be a protocol for communication between the subprocess and the main application. For a DIY solution, twisted's NetstringReceiver may be a useful starting point.

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.

See *the events_twisted module* page for more information about *soco.events_twisted*.

1.8.2 Example: setting up

1.8.2.1 soco.events

```
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
```

1.8.2.2 soco.events_twisted

```
import soco
from soco import events_twisted
soco.config.EVENTS_MODULE = events_twisted
from twisted.internet import reactor
import logging
logging.basicConfig(level=logging.DEBUG)

def print_event(event):
    print (event)
    print(event.sid)
    print(event.seq)
```

(continues on next page)

(continued from previous page)

```
def main():
    # pick a device
    device = soco.discover().pop()
    # Subscribe to ZGT events
    sub = device.zoneGroupTopology.subscribe().subscription
    # print out the events as they arise
    sub.callback = print_event

    def before_shutdown():
        sub.unsubscribe()
        events_twisted.event_listener.stop()

    reactor.addSystemEventTrigger(
        'before', 'shutdown', before_shutdown)

if __name__=='__main__':
    reactor.callWhenRunning(main)
    reactor.run()
```

1.8.3 Examples: specific features

1.8.3.1 Autorenewal

A Subscription may be granted by the Sonos system for a finite time. Unless it is renewed before it times out, the subscription will become defunct once it times out. To avoid this, the autorenewal feature can be used. If the auto-renew flag is set to True, the subscription will automatically renew when 85% of its time has expired.

soco.events:

```
sub = device.renderingControl.subscribe(auto_renew=True)
```

soco.events_twisted:

```
sub = device.renderingControl.subscribe(auto_renew=True).subscription
```

1.8.3.2 Timeout

When subscribing for events, a timeout of a specific duration can be requested.

soco.events:

```
sub = device.renderingControl.subscribe(requested_timeout=60) # 60 seconds
```

soco.events_twisted:

```
sub = device.renderingControl.subscribe(requested_timeout=60).subscription
```

1.8.3.3 Renewal

To renew without relying on autorenewal, the renew method can be used:

```
sub.renew(requested_timeout=10)
```

1.8.3.4 Autorenew failure

If you want your application to respond to an autorenew failure (for example if the Sonos system dropped off the network), you can set an optional callback that will be called with the exception that occurred on the attempted autorenew:

```
import logging
logging.basicConfig()
log = logging.getLogger(__name__)

def errback(exception): # events_twisted: failure
    msg = 'Error received on autorenew: {}'.format(str(exception))
    # Redundant, as the exception will be logged by the events module
    log.exception(msg)

sub.auto_renew_fail=errback
```

Note: In *soco.events* the auto_renew_fail function will be called from a thread, so it must be threadsafe.

1.8.3.5 Lenient error handling

By default, if an exception occurs when subscribing, renewing or unsubscribing a subscription, the exception will be raised. This can be changed so the exception is logged instead, by setting the strict flag to be false:

```
sub.unsubscribe(strict=False)
```

1.8.3.6 Events_twisted: adding callbacks and errbacks

If the events_twisted module is used, subscribe, renew and unsubscribe return a Deferred, the result of which will be the *Subscription* instance. Callbacks and errbacks can be added in the usual way:

```
device.renderingControl.subscribe().addCallback(myCallback).addErrback(
    myErrback)
```

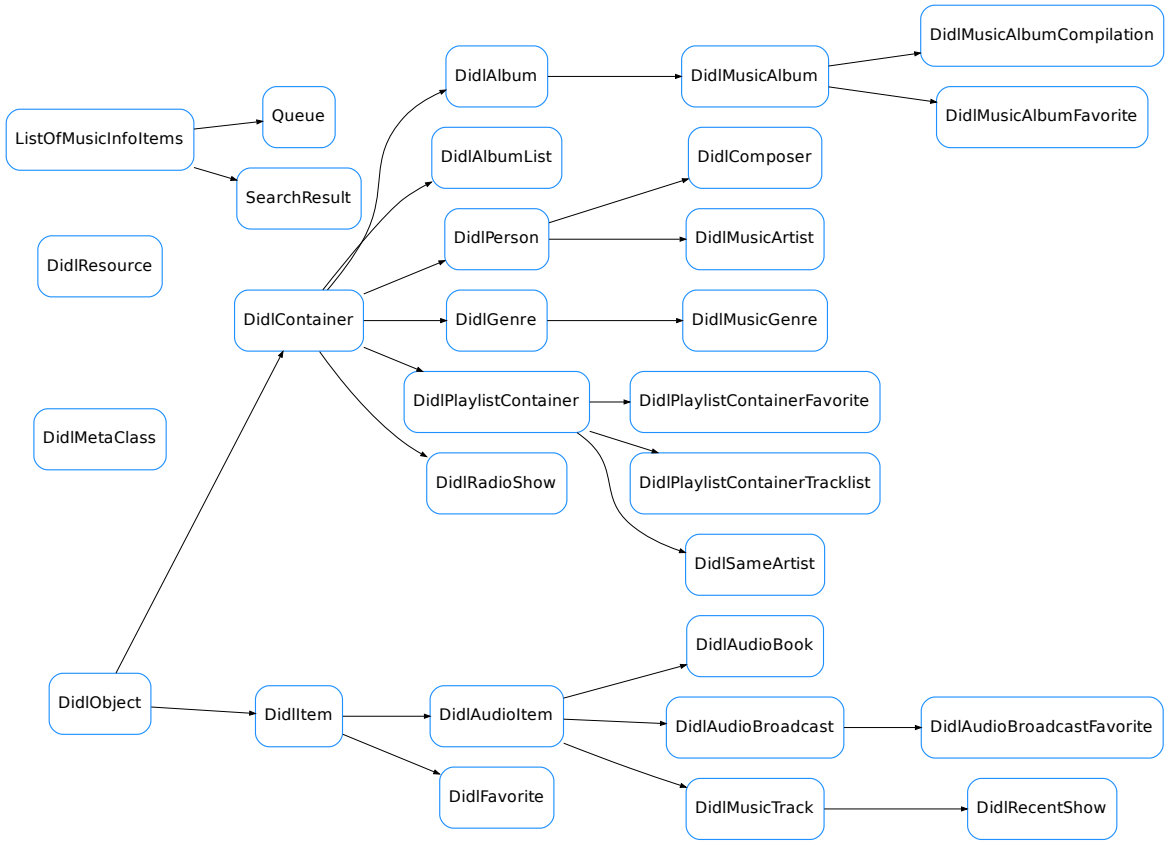
1.9 The Music Library Data Structures

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`.

1.10 soco package

1.10.1 Subpackages

1.10.1.1 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
A unique identifier for the music service to which this account relates, eg `'2311'` for Spotify.

Type str

serial_number = None
A unique identifier for this account

Type str

nickname = None
The account's nickname

Type str

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

Type bool

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

Type str

metadata = None
Metadata for the account

Type str

oa_device_id = None
Used for OpenAuth id for some services

Type str

key = None
Used for OpenAuthid for some services

Type str

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"
```

(continues on next page)

(continued from previous page)

```
        },
 ...
    ]
}
```

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
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')
```

Type list

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
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.

Type str

get_metadata(*item='root'*, *index=0*, *count=100*, *recursive=False*)
Get metadata for a container or item.

Parameters

- **item** (*str or MusicServiceItem*) – The container or item to browse given either as a MusicServiceItem instance or as a str. 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&sn=4'`

Returns

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

Return type str

1.10.1.2 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__.

name

human-readable name of the plugin

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.

1.10.2 Submodules

1.10.2.1 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
```

(continues on next page)

(continued from previous page)

```
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
The alarm’s start time.

Type datetime.time

duration = None
The alarm’s duration.

Type datetime.time

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

Type bool

program_uri = None

program_metadata = None
The uri to play.

Type str

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

Type bool

play_mode
The play mode for the alarm.

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

Type str

volume
The alarm’s volume (0-100).

Type int

recurrence
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.

Type str

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.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.

1.10.2.2 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
whether the cache is enabled

Type bool

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

The default caching expiry interval in seconds.

Type int

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.

1.10.2.3 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
```

1.10.2.4 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**
alias of *soco.core.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_ADVERTISE_IP = None**
The IP on which to advertise to Sonos.

The default of None means that the relevant IP address will be detected automatically.

See also:

The *soco.events_base* module.

soco.config.**EVENT_LISTENER_IP = None**
The IP on which the event listener listens.

The default of None means that the relevant IP address will be detected automatically.

See also:

The *soco.events_base* 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_base* module.

soco.config.**EVENTS_MODULE = <module 'soco.events' from '/home/docs/checkouts/readthedocs.o**
The events module to be used by the *soco.services* module.

The default of None means the *soco.events* module will be used.

See also:

The *soco.events* and *soco.events_twisted* modules.

1.10.2.5 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.

Basic Methods

play_from_queue(index[, start])	Play a track from the queue by index.
play()	Play the currently selected track.
play_uri([uri, meta, title, start, force_radio])	Play a URI.
pause()	Pause the currently playing track.
stop()	Stop the currently playing track.
seek(timestamp)	Seek to a given timestamp in the current track, specified in the format of HH:MM:SS or H:MM:SS.
next()	Go to the next track.
previous()	Go back to the previously played track.
mute	The speaker's mute state.
volume	The speaker's volume.
play_mode	The queue's play mode.
cross_fade	The speaker's cross fade state.
ramp_to_volume(volume[, ramp_type])	Smoothly change the volume.
set_relative_volume(relative_volume)	Adjust the volume up or down by a relative amount.
get_current_track_info()	Get information about the currently playing track.
get_speaker_info([refresh, timeout])	Get information about the Sonos speaker.
get_current_transport_info()	Get the current playback state.

Queue Management

get_queue([start, max_items, full_album_art_uri])	Get information about the queue.
queue_size	Size of the queue.
add_to_queue(queueable_item[, position, as_next])	Add a queueable item to the queue.
add_uri_to_queue(uri[, position, as_next])	Add the URI to the queue.
add_multiple_to_queue(items[, container])	Add a sequence of items to the queue.
remove_from_queue(index)	Remove a track from the queue by index.
clear_queue()	Remove all tracks from the queue.

Group Management

group	The Zone Group of which this device is a member.
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.
all_groups	All available groups.
all_zones	All available zones.
visible_zones	All visible zones.

Player Identity and Settings

player_name	The speaker's name.
uid	A unique identifier.
household_id	A unique identifier for all players in a household.
is_visible	Is this zone visible?
is_bridge	Is this zone a bridge?
is_coordinator	Is this zone a group coordinator?
is_soundbar	Is this zone a soundbar (i.e.
bass	The speaker's bass EQ.
treble	The speaker's treble EQ.
loudness	The speaker's loudness compensation.
balance	The left/right balance for the speaker(s).
night_mode	The speaker's night mode.
dialog_mode	The speaker's dialog mode.
supports_fixed_volume	Whether the device supports fixed volume output.
fixed_volume	The device's fixed volume output setting.
trueplay	Whether Trueplay is enabled on this device.
status_light	The white Sonos status light between the mute button and the volume up button on the speaker.
buttons_enabled	Whether the control buttons on the device are enabled.

Playlists and Favorites

get_sonos_playlists(*args, **kwargs)	Convenience method for calling `soco.music_library.get_music_library_information('sonos_playlists')`
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.
reorder_sonos_playlist(sonos_playlist, ...)	Reorder and/or Remove tracks in a Sonos playlist.
clear_sonos_playlist(sonos_playlist[, update_id])	Clear all tracks from a Sonos playlist.
move_in_sonos_playlist(sonos_playlist, ...)	Move a track to a new position within a Sonos Playlist.
remove_from_sonos_playlist(sonos_playlist, track)	Remove a track from a Sonos Playlist.
get_sonos_playlist_by_attr(attr_name, match)	Return the first Sonos Playlist DidlPlaylistContainer that matches the attribute specified.
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.

Miscellaneous

music_source	The current music source (radio, TV, line-in, etc.).
music_source_from_uri(uri)	Determine a music source from a URI.
is_playing_radio	Is the speaker playing radio?
is_playing_tv	Is the playbar speaker input from TV?
is_playing_line_in	Is the speaker playing line-in?
switch_to_line_in([source])	Switch the speaker's input to line-in.
switch_to_tv()	Switch the playbar speaker's input to TV.
set_sleep_timer(sleep_time_seconds)	Sets the sleep timer.
get_sleep_timer()	Retrieves remaining sleep time, if any
create_stereo_pair(rh_slave_speaker)	Create a stereo pair.
separate_stereo_pair()	Separate a stereo pair.
get_battery_info([timeout])	Get battery information for a Sonos speaker.

Warning: Properties on this object 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.

Note: Since all methods/properties on this object will result in an UPnP request, they might result in an exception without it being mentioned in the Raises section.

In most cases, the exception will be a *soco.exceptions.SoCoUPnPException* (if the player returns an UPnP error code), but in special cases it might also be another *soco.exceptions.SoCoException* or even a `requests` exception.

ip_address = None
The speaker's ip address

player_name
The speaker's name.

Type str

uid
A unique identifier.

Looks like: `'RINCON_000XXXXXXXXX1400'`

Type str

household_id
A unique identifier for all players in a household.

Looks like: `'Sonos_asahHKgjgJGjgjGjggjJgjJG34'`

Type str

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.

Type bool

is_bridge
Is this zone a bridge?

Type bool

is_coordinator
Is this zone a group coordinator?

Type bool

is_soundbar
Is this zone a soundbar (i.e. has night mode etc.)?

Type bool

play_mode
The queue's play mode.

Case-insensitive options are:

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

Type str

cross_fade
The speaker's cross fade state.

True if enabled, False otherwise

Type bool

ramp_to_volume(*volume*, *ramp_type='SLEEP_TIMER_RAMP_TYPE'*)
Smoothly change the volume.

There are three ramp types available:

- `'SLEEP_TIMER_RAMP_TYPE'` (default): Linear ramp from the current volume up or down to the new volume. The ramp rate is 1.25 steps per second. For example: To change from volume 50 to volume 30 would take 16 seconds.
- `'ALARM_RAMP_TYPE'`: Resets the volume to zero, waits for about 30 seconds, and then ramps the volume up to the desired value at a rate of 2.5 steps per second. For example: Volume 30 would take 12 seconds for the ramp up (not considering the wait time).
- `'AUTOPLAY_RAMP_TYPE'`: Resets the volume to zero and then quickly ramps up at a rate of 50 steps per second. For example: Volume 30 will take only 0.6 seconds.

The ramp rate is selected by Sonos based on the chosen ramp type and the resulting transition time returned. This method is non blocking and has no network overhead once sent.

Parameters

- **volume** (*int*) – The new volume.
- **ramp_type** (*str, optional*) – The desired ramp type, as described above.

Returns The ramp time in seconds, rounded down. Note that this does not include the wait time.

Return type int

set_relative_volume(*relative_volume*)
Adjust the volume up or down by a relative amount.

If the adjustment causes the volume to overshoot the maximum value of 100, the volume will be set to 100. If the adjustment causes the volume to undershoot the minimum value of 0, the volume will be set to 0.

Note that this method is an alternative to using addition and subtraction assignment operators (+=, -=) on the *volume* property of a *SoCo* instance. These operators perform the same function as *set_relative_volume* but require two network calls per operation instead of one.

Parameters **relative_volume** (*int*) – The relative volume adjustment. Can be positive or negative.

Returns The new volume setting.

Return type int

Raises ValueError – If `relative_volume` cannot be cast as an integer.

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.

Parameters

- **index** (*int*) – 0-based index of the track to play
- **start** (*bool*) – If the item that has been set should start playing

play()
Play the currently selected track.

play_uri(*uri=''*, *meta=''*, *title=''*, *start=True*, *force_radio=False*)
Play a URI.

Playing a URI will replace what was playing with the stream given by the URI. For some streams at least a title is required as metadata. This can be provided using the `meta` argument or the `title` argument. If the `title` argument is provided minimal metadata will be generated. If `meta` argument is provided the `title` argument is ignored.

Parameters

- **uri** (*str*) – URI of the stream to be played.
- **meta** (*str*) – The metadata to show in the player, DIDL format.
- **title** (*str*) – The title to show in the player (if no meta).
- **start** (*bool*) – If the URI that has been set should start playing.
- **force_radio** (*bool*) – forces a uri to play as a radio stream.

On a Sonos controller music is shown with one of the following display formats and controls:

- Radio format: Shows the name of the radio station and other available data. No seek, next, previous, or voting capability. Examples: TuneIn, radioPup
- Smart Radio: Shows track name, artist, and album. Limited seek, next and sometimes voting capability depending on the Music Service. Examples: Amazon Prime Stations, Pandora Radio Stations.
- Track format: Shows track name, artist, and album the same as when playing from a queue. Full seek, next and previous capabilities. Examples: Spotify, Napster, Rhapsody.

How it is displayed is determined by the URI prefix: `x-sonosapi-stream:`, `x-sonosapi-radio:`, `x-rincon-mp3radio:`, `hls-radio:` default to radio or smart radio format depending on the stream. Others default to track format: `x-file-cifs:`, `aac:`, `http:`, `https:`, `x-sonos-spotify:` (used by Spotify), `x-sonosapi-hls-static:` (Amazon Prime), `x-sonos-http:` (Google Play & Napster).

Some URIs that default to track format could be radio streams, typically `http:`, `https:` or `aac:`. To force display and controls to Radio format set `force_radio=True`

Note: Other URI prefixes exist but are less common. If you have information on these please add to this doc string.

Note: A change in Sonos® (as of at least version 6.4.2) means that the devices no longer accepts ordinary `http:` and `https:` URIs for radio stations. This method has the option to replaces these prefixes with the one that Sonos® expects: `x-rincon-mp3radio:` by using the "force_radio=True" parameter. A few streams may fail if not forced to to Radio format.

---

pause()
Pause the currently playing track.

stop()
Stop the currently playing track.

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

Raises `ValueError` – if the given timestamp is invalid.

next()
Go to the next track.

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.

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.

Type bool

volume
The speaker's volume.

An integer between 0 and 100.

Type int

bass
The speaker's bass EQ.

An integer between -10 and 10.

Type int

treble
The speaker's treble EQ.

An integer between -10 and 10.

Type int

loudness
The speaker's loudness compensation.

True if on, False otherwise.

Loudness is a complicated topic. You can read about it on Wikipedia: https://en.wikipedia.org/wiki/Loudness

Type bool

balance
The left/right balance for the speaker(s).

Returns A 2-tuple (left_channel, right_channel) of integers between 0 and 100, representing the volume of each channel. E.g., (100, 100) represents full volume to both channels, whereas (100, 0) represents left channel at full volume, right channel at zero volume.

Return type tuple

night_mode
The speaker's night mode.

True if on, False if off, None if not supported.

Type bool

dialog_mode
The speaker's dialog mode.

True if on, False if off, None if not supported.

Type bool

trueplay
Whether Trueplay is enabled on this device. True if on, False if off.

Devices that do not support Trueplay, or which do not have a current Trueplay calibration, will return `None` on getting the property, and raise a *NotSupportedException* when setting the property.

Can only be set on visible devices. Attempting to set on non-visible devices will raise a *SoCoNotVisibleException*.

Type bool

supports_fixed_volume
Whether the device supports fixed volume output.

Type bool

fixed_volume
The device's fixed volume output setting.

True if on, False if off. Only applicable to certain Sonos devices (Connect and Port at the time of writing). All other devices always return False.

Attempting to set this property for a non-applicable device will raise a *NotSupportedException*.

Type bool

all_groups
All available groups.

Type set of *soco.groups.ZoneGroup*

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

None if this zone is a slave in a stereo pair.

Type *soco.groups.ZoneGroup*

all_zones
All available zones.

Type set of *soco.groups.ZoneGroup*

visible_zones
All visible zones.

Type set of *soco.groups.ZoneGroup*

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.

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.

create_stereo_pair(*rh_slave_speaker*)
Create a stereo pair.

This speaker becomes the master, left-hand speaker of the stereo pair. The `rh_slave_speaker` becomes the right-hand speaker. Note that this operation will succeed on dissimilar speakers, unlike when using the official Sonos apps.

Parameters **rh_slave_speaker** (SoCo) – The speaker that will be added as the right-hand, slave speaker of the stereo pair.

Raises `SoCoUPnPException` – if either speaker is already part of a stereo pair.

separate_stereo_pair()
Separate a stereo pair.

This can be called on either the master (left-hand) speaker, or on the slave (right-hand) speaker, to create two independent zones.

Raises `SoCoUPnPException` – if the speaker is not a member of a stereo pair.

switch_to_line_in(*source=None*)
Switch the speaker's input to line-in.

Parameters **source** (SoCo) – The speaker whose line-in should be played. Default is line-in from the speaker itself.

is_playing_radio
Is the speaker playing radio?

Type bool

is_playing_line_in
Is the speaker playing line-in?

Type bool

is_playing_tv
Is the playbar speaker input from TV?

Type bool

static music_source_from_uri(*uri*)
Determine a music source from a URI.

Parameters **uri** (*str*) – The URI representing the music source

Returns The current source of music.

Return type str

Possible return values are:

- `'NONE'` – speaker has no music to play.
- `'LIBRARY'` – speaker is playing queued titles from the music library.
- `'RADIO'` – speaker is playing radio.
- `'WEB_FILE'` – speaker is playing a music file via http/https.
- `'LINE_IN'` – speaker is playing music from line-in.
- `'TV'` – speaker is playing input from TV.
- `'AIRPLAY'` – speaker is playing from AirPlay.
- `'UNKNOWN'` – any other input.

The strings above can be imported as `MUSIC_SRC_LIBRARY`, `MUSIC_SRC_RADIO`, etc.

music_source
The current music source (radio, TV, line-in, etc.).

Possible return values are the same as used in *music_source_from_uri()*.

Type str

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

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

True if on, otherwise False.

Type bool

buttons_enabled
Whether the control buttons on the device are enabled.

`True` if the control buttons are enabled, `False` if disabled.

This property can only be set on visible speakers, and will enable or disable the buttons for all speakers in any bonded set (e.g., a stereo pair). Attempting to set it on invisible speakers (e.g., the RH speaker of a stereo pair) will raise a *SoCoNotVisibleException*.

Type bool

get_current_track_info()
Get information about the currently playing track.

Returns A dictionary containing information about the currently playing track: playlist_position, duration, title, artist, album, position and an album_art link.

Return type dict

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.

Parameters

- **refresh** (*bool*) – Refresh the speaker info cache.
- **timeout** – How long to wait for the server to send 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.

Return type dict

get_current_transport_info()
Get the current playback state.

Returns

The following information about the speaker's playing state:

- current_transport_state (`PLAYING`, `TRANSITIONING`, `PAUSED_PLAYBACK`, `STOPPED`)
- current_transport_status (OK, ?)
- current_speed(1, ?)

Return type dict

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
Size of the queue.

Type int

get_sonos_playlists(**args*, ***kwargs*)
Convenience method for calling `soco.music_library.get_music_library_information('sonos_playl`

Refer to the docstring for that method: *get_music_library_information*

add_uri_to_queue(*uri*, *position=0*, *as_next=False*)
Add the URI to the queue.

For arguments and return value see *add_to_queue*.

add_to_queue(*queueable_item*, *position=0*, *as_next=False*)
Add a queueable item to the queue.

Parameters

- **queueable_item** (*DidlObject or MusicServiceItem*) – The item to be added to the queue
- **position** (*int*) – The index (1-based) at which the URI should be added. Default is 0 (add URI at the end of the queue).
- **as_next** (*bool*) – Whether this URI should be played as the next track in shuffle mode. This only works if `play_mode=SHUFFLE`.

Returns The index of the new item in the queue.

Return type int

add_multiple_to_queue(*items*, *container=None*)
Add a sequence of items to the queue.

Parameters

- **items** (*list*) – A sequence of items to the be added to the queue
- **container** (*DidlObject, optional*) – A container object which includes the items.

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.

Parameters **index** (*int*) – The (0-based) index of the track to remove

clear_queue()
Remove all tracks from the queue.

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

Returns: dict: A dictionary 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.

Deprecated since version 0.13: Will be removed in version 0.15. Use `soco.music_library.get_favorite_radio_shows` instead.

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

See *get_favorite_radio_shows()* for return type and remarks.

Deprecated since version 0.13: Will be removed in version 0.15. Use `soco.music_library.get_favorite_radio_stations` instead.

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

See *get_favorite_radio_shows()* for return type and remarks.

Deprecated since version 0.13: Will be removed in version 0.15. Use `soco.music_library.get_sonos_favorites` instead.

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

Parameters **title** – Name of the playlist

Return type *DidlPlaylistContainer*

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

Parameters **title** – Name of the playlist

Return type *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** (DidlObject) – the item to add to the Sonos' playlist
- **sonos_playlist** (DidlPlaylistContainer) – the Sonos' playlist to which the item should be added

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

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')
```

get_battery_info(*timeout=3.0*)
Get battery information for a Sonos speaker.

Obtains battery information for Sonos speakers that report it. This only applies to Sonos Move speakers at the time of writing.

This method may only work on Sonos 'S2' systems.

Parameters **timeout** (*float, optional*) – The timeout to use when making the HTTP request.

Returns

A dict containing battery status data.

Example return value:

```
{'Health': 'GREEN',
 'Level': 100,
 'Temperature': 'NORMAL',
 'PowerSource': 'SONOS_CHARGING_RING'}
```

Return type dict

Raises

- NotSupportedException – If the speaker does not report battery information.
- ConnectionError – If the HTTP connection failed, or returned an unsuccessful status code.
- TimeoutError – If making the HTTP connection, or reading the response, timed out.

1.10.2.6 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.**didl_class_to_soco_class**(*didl_class*)
Translate a DIDL-Lite class to the corresponding SoCo data structures class

soco.data_structures.**form_name**(*didl_class*)
Return an improvised name for vendor extended classes

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
a percent encoded URI

Type (str)

protocol_info = None
protocol information.

Type (str)

duration = None
playback duration

Type str

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*.

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

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*.

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

get_uri(*resource_nr=0*)
Return the uri to use for playing this item.

Parameters **resource_nr** (*int*) – The index of the resource. Note that there is no known object with more than one resource, so you can probably keep the default value (0).

Returns The uri.

Return type str

set_uri(*uri*, *resource_nr=0*, *protocol_info=None*)
Set a resource uri for this instance. If no resource exists, create a new one with the given protocol info.

Parameters

- **uri** (*str*) – The resource uri.
- **resource_nr** (*int*) – The index of the resource on which to set the uri. If it does not exist, a new resource is added to the list. Note that by default, only the uri of the first resource is used for playing the item.
- **protocol_info** (*str*) – Protocol info for the resource. If none is given and the resource does not exist yet, a default protocol info is constructed as `'[uri prefix]:*:*:*'`.

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 = {'album_art_uri': ('upnp', 'albumArtURI'), '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.**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 = {'album_art_uri': ('upnp', 'albumArtURI'), '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.**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 = {'album': ('upnp', 'album'), 'album_art_uri': ('upnp', 'albumArtURI')
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.**DidlAudioBook**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)

Class that represents an audio book.

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.audioBook'
str - the DIDL Lite class for this object.

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

_translation = {'album_art_uri': ('upnp', 'albumArtURI'), 'contributor': ('dc', 'con
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 = {'album_art_uri': ('upnp', 'albumArtURI'), 'channel_nr': ('upnp', 'cha
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.**DidlRecentShow**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)

Class that represents a recent radio show/podcast.

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.recentShow'

str - the DIDL Lite class for this object.

tag = 'item'

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

_translation = {'album': ('upnp', 'album'), 'album_art_uri': ('upnp', 'albumArtURI')

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 = {'album_art_uri': ('upnp', 'albumArtURI'), 'channel_nr': ('upnp', 'ch
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.**DidlFavorite**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)

Class that represents a Sonos favorite.

Note that the favorite itself isn't playable in all cases, please use the object returned by `favorite.reference` instead.

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.itemobject.item.sonos-favorite'
str - the DIDL Lite class for this object.

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

_translation = {'album_art_uri': ('upnp', 'albumArtURI'), '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.

reference
The Didl object this favorite refers to.

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 = {'contributor': ('dc', 'contributor'), '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.**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 = {'album_art_uri': ('upnp', 'albumArtURI'), 'artist': ('upnp', 'artist
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 = {'album_art_uri': ('upnp', 'albumArtURI'), 'artist': ('upnp', 'artist
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 = {'album_art_uri': ('upnp', 'albumArtURI'), 'artist': ('upnp', 'artist
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
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
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'), '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.**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 = {'artist': ('upnp', 'artist'), 'contributor': ('dc', 'contributor'),
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.**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 = {'artist': ('upnp', 'artist'), 'contributor': ('dc', 'contributor'),
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.**DidlPlaylistContainerFavorite**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)

Class that represents a Sonos favorite 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.sonos-favorite'
str - the DIDL Lite class for this object.

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

_translation = {'artist': ('upnp', 'artist'), 'contributor': ('dc', 'contributor'),
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.**DidlPlaylistContainerTracklist**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)

Class that represents a Sonos tracklist.

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.tracklist'
str - the DIDL Lite class for this object.

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

_translation = {'artist': ('upnp', 'artist'), 'contributor': ('dc', 'contributor'),
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 = {'creator': ('dc', 'creator'), 'description': ('dc', 'description'),
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 = {'creator': ('dc', 'creator'), 'description': ('dc', 'description'),
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.**DidlRadioShow**(*title*, *parent_id*, *item_id*, *restricted=True*, *resources=None*, *desc='RINCON_AssociatedZPUDN'*, ***kwargs*)

Class that represents a radio show.

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.radioShow'
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.**ListOfMusicInfoItems**(*items*, *number_returned*, *total_matches*, *update_id*)

Abstract container class for a list of music information items.

Instances of this class are returned from queries into the music library or to music services. The attributes *total_matches* and *number_returned* are used to ascertain whether paging is required in order to retrive all elements of the query. *total_matches* is the total number of results to the query and *number_returned* is the number of results actually returned. If the two differ, paging is required. Paging is typically performed with the `start` and `max_items` arguments to the query method. See e.g. the *get_music_library_information()* method for details.

number_returned
the number of returned matches.

Type str

total_matches
the number of total matches.

Type str

update_id
the update ID.

Type str

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
the search type.

Type str

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

1.10.2.7 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*, *allow_network_scan=False*, ***network_scan_kwargs*)
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, 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(s) for UDP multicast messages will be used. This is probably what you want to happen. Defaults to `None`.
- **allow_network_scan** (*bool, optional*) – If normal discovery fails, fall back to a scan of the attached network(s) to detect Sonos devices.
- ****network_scan_kwargs** – Arguments for the *scan_network* function. See its docstring for details.

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

Return type set

soco.discovery.**any_soco**(*allow_network_scan=False*, ***network_scan_kwargs*)
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.

Parameters

- **allow_network_scan** (*bool, optional*) – If normal discovery fails, fall back to a scan of the attached network(s) to detect Sonos devices.
- ****network_scan_kwargs** – Arguments for the *scan_network* function. See its docstring for details.

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

Return type *SoCo*

soco.discovery.**by_name**(*name*, *allow_network_scan=False*, ***network_scan_kwargs*)
Return a device by name.

Parameters

- **name** (*str*) – The name of the device to return.
- **allow_network_scan** (*bool, optional*) – If normal discovery fails, fall back to a scan of the attached network(s) to detect Sonos devices.
- ****network_scan_kwargs** – Arguments for the *scan_network* function. See its docstring for details.

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

Return type *SoCo*

soco.discovery.**scan_network**(*include_invisible=False*, *multi_household=False*, *max_threads=256*, *scan_timeout=0.1*, *min_netmask=24*, *networks_to_scan=None*)
Scan all attached networks for Sonos devices.

This function scans the IPv4 networks to which this node is attached, searching for Sonos devices. Multiple parallel threads are used to scan IP addresses in parallel for faster discovery.

Public, loopback and link local IP ranges are excluded from the scan, and the scope of the search can be controlled by setting a minimum netmask.

Alternatively, a list of networks to scan can be provided.

This function is intended for use when the usual discovery function is not working, perhaps due to multicast problems on the network to which the SoCo host is attached. The function can also be used to find a complete list of speakers when there are multiple Sonos households present. For example, this is the case where there are 'split' S1/S2 Sonos systems on the network.

Note that this call may fail to find speakers present on the network, and this can be due to ARP cache misses and ARP requests that don't complete within the timeout. The call can be retried with longer values for scan_timeout if necessary.

Parameters

- **include_invisible** (*bool, optional*) – Whether to include invisible Sonos devices in the set of devices returned.
- **multi_household** (*bool, optional*) – Whether to find all the speakers on the network exhaustively. If set to False, discovery will stop as soon as at least one speaker is

found. In the case of multiple households on the attached networks, this means that only the speakers from the first-discovered household will be returned. If set to `True`, discovery will proceed until all speakers, from all households, have been found.

- **max_threads** (*int, optional*) – The maximum number of threads to use when scanning the network.
- **scan_timeout** (*float, optional*) – The network timeout in seconds to use when checking each IP address for a Sonos device.
- **min_netmask** (*int, optional*) – The minimum number of netmask bits. Used to constrain the network search space.
- **networks_to_scan** (*list, optional*) – A `list` of IPv4 networks to search, each a `str` of form "192.168.0.1/24". Only the specified networks will be searched. The 'min_netmask' option (if supplied) is ignored.

Returns A set of *SoCo* instances, one for each zone found, or else `None`.

Return type set

soco.discovery.**scan_network_by_household_id**(*household_id*, *include_invisible=False*, ***network_scan_kwargs*)

Convenience function to find the zones in a specific Sonos household.

Parameters

- **household_id** (*str*) – The Sonos household ID to search for. IDs take the form 'Sonos_XXXXXXXXXXXXXXXXXXXXXXXXXX'.
- **include_invisible** (*bool, optional*) – Whether to include invisible Sonos devices in the set of devices returned.
- ****network_scan_kwargs** – Arguments for the *scan_network* function. See its docstring for details. (Note that the argument 'multi_household' is forced to `True` when this function is called.)

Returns A set of *SoCo* instances, one for each zone found, or else `None`.

Return type set

soco.discovery.**scan_network_get_household_ids**(***network_scan_kwargs*)

Convenience function to find the all Sonos households on the attached networks.

Parameters ****network_scan_kwargs** – Arguments for the *scan_network* function. See its docstring for details. (Note that the argument 'multi_household' is forced to `True` when this function is called.)

Returns A set of Sonos household IDs, each in the form of a `str` like 'Sonos_XXXXXXXXXXXXXXXXXXXXXXXXXX'.

Return type set

soco.discovery.**scan_network_get_by_name**(*name*, *household_id=None*, ***network_scan_kwargs*)

Convenience function to use *scan_network* to find a zone by its name.

Note that if there are multiple zones with the same name, then only one of the zones will be returned. Optionally, the search can be constrained to a specific household.

Parameters

- **name** (*str*) – The name of the zone to find.

- **household_id** (*str, optional*) – Use this to find the zone in a specific Sonos household.
- ****network_scan_kwargs** – Arguments for the *scan_network* function. See its docstring for details. (Note that the argument 'multi_household' is forced to `True` when this function is called.)

Returns A *SoCo* instance representing the zone, or `None` if no matching zone is found. Only returns visible zones.

Return type *SoCo*

soco.discovery.**scan_network_any_soco**(*household_id=None*, ***network_scan_kwargs*)

Convenience function to use *scan_network* to find any zone, optionally specifying a Sonos household.

Parameters

- **household_id** (*str, optional*) – Use this to find a zone in a specific Sonos household.
- ****network_scan_kwargs** – Arguments for the *scan_network* function. See its docstring for details.

Returns A *SoCo* instance representing the zone, or `None` if no zone is found (or no zone is found that matches a supplied household_id).

Return type *SoCo*

1.10.2.8 soco.events module

Classes to handle Sonos UPnP Events and Subscriptions.

The *Subscription* class from this module will be used in *soco.services* unless *config.EVENTS_MODULE* is set to point to *soco.events_twisted*, in which case *soco.events_twisted.Subscription* will be used. See the Example in *soco.events_twisted*.

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 logging
logging.basicConfig()
import soco
from pprint import pprint
from soco.events import event_listener
# pick a device at random and use it to get
# the group coordinator
device = soco.discover().pop().group.coordinator
print (device.player_name)
sub = device.renderingControl.subscribe()
sub2 = device.avTransport.subscribe()
```

(continues on next page)

(continued from previous page)

```
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.**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**(**args*, ***kwargs*)

Handles HTTP NOTIFY Verbs sent to the listener server. Inherits from *soco.events_base.EventNotifyHandlerBase*.

do_NOTIFY()

Serve a NOTIFY request by calling *handle_notification* with the headers and content.

log_message(*fmt*, **args*)

Log an arbitrary message.

This is used by all other logging functions. Override it if you have specific logging wishes.

The first argument, FORMAT, is a format string for the message to be logged. If the format string contains any % escapes requiring parameters, they should be specified as subsequent arguments (it's just like printf!).

The client ip and current date/time are prefixed to every message.

class soco.events.**EventServerThread**(*server*)

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

Used to signal that the server should stop.

Type threading.Event

server = None

The (ip, port) address on which the server is configured to listen.

Type tuple

run()

Start the server

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

stop()
Stop the server.

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

Runs an http server in a thread which is an endpoint for NOTIFY requests from Sonos devices. Inherits from *soco.events_base.EventListenerBase*.

listen(*ip_address*)
Start the event listener listening on the local machine at port 1400 (default). If this port is unavailable, the listener will attempt to listen on the next available port, within a range of 100.

Make sure that your firewall allows connections to this port.

This method is called by *soco.events_base.EventListenerBase.start*

Parameters **ip_address** (*str*) – The local network interface on which the server should start listening.

Returns *requested_port_number*. Included for compatibility with *soco.events_twisted.EventListener.listen*

Return type int

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

stop_listening(*address*)
Stop the listener.

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

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.

subscribe(*requested_timeout=None*, *auto_renew=False*, *strict=True*)
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.

This method calls *events_base.SubscriptionBase.subscribe*.

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.

- **strict** (*bool, optional*) – If True and an Exception occurs during execution, the Exception will be raised or, if False, the Exception will be logged and the Subscription instance will be returned. Default `True`.

Returns The Subscription instance.

Return type *Subscription*

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.

This method calls *events_base.SubscriptionBase.renew*.

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.
- **is_autorenew** (*bool, optional*) – Whether this is an autorenewal. Default 'False'.
- **strict** (*bool, optional*) – If True and an Exception occurs during execution, the Exception will be raised or, if False, the Exception will be logged and the Subscription instance will be returned. Default `True`.

Returns The Subscription instance.

Return type *Subscription*

unsubscribe()
Unsubscribe from the service's events. Once unsubscribed, a Subscription instance should not be reused

This method calls *events_base.SubscriptionBase.unsubscribe*.

Parameters **strict** (*bool, optional*) – If True and an Exception occurs during execution, the Exception will be raised or, if False, the Exception will be logged and the Subscription instance will be returned. Default `True`.

Returns The Subscription instance.

Return type *Subscription*

1.10.2.9 soco.events_base module

Base classes used by *soco.events* and *soco.events_twisted*.

soco.events_base.**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).
- a *SoCoFault* (if a variable contains illegal metadata)

Return type dict

class soco.events_base.**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`. The values may be *SoCoFault* objects if the metadata could not be parsed.

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'
```

class soco.events_base.**EventNotifyHandlerBase**

Base class for *soco.events.EventNotifyHandler* and *soco.events_twisted.EventNotifyHandler*.

handle_notification(*headers*, *content*)

Handle a `NOTIFY` request by building an *Event* object and sending it to the relevant Subscription object.

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

Parameters

- **headers** (*dict*) – A dict of received headers.
- **content** (*str*) – A string of received content.

Note: Each of the *soco.events* and the *soco.events_twisted* modules has a **subscriptions_map** object which keeps a record of Subscription objects. The *get_subscription* method of the **subscriptions_map** object is used to look up the subscription to which the event relates. When the Event Listener runs in a thread (the default), a lock is used by this method for thread safety. The *send_event* method of the relevant Subscription will first check to see whether the *callback* variable of the Subscription has been set. If it has been and is callable, then the *callback* will be called with the *Event* object. Otherwise, the *Event* object will be sent to the event queue of the Subscription object. The *callback* variable of the Subscription object is intended for use only if *soco.events_twisted* is being used, as calls to it are not threadsafe.

This method calls the log_event method, which must be overridden in the class that inherits from this class.

class soco.events_base.**EventListenerBase**

Base class for *soco.events.EventListener* and *soco.events_twisted.EventListener*.

is_running = None

Indicates whether the server is currently running

Type bool

requested_port_number = None

Port on which to listen.

Type int

start(*any_zone*)

Start the event listener listening on the local machine.

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.

stop()

Stop the Event Listener.

listen(*ip_address*)

Start the event listener listening on the local machine. This method is called by *start*.

Parameters **ip_address** (*str*) – The local network interface on which the server should start listening.

Returns The port on which the server is listening.

Return type int

Note: This method must be overridden in the class that inherits from this class.

stop_listening(*address*)

Stop the listener.

Note: This method must be overridden in the class that inherits from this class.

class soco.events_base.**SubscriptionBase**(*service*, *event_queue=None*)

Base class for *soco.events.Subscription* and *soco.events_twisted.Subscription*

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

A unique ID for this subscription

Type str

timeout = None

The amount of time in seconds until the subscription expires.

Type int

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

Type bool

events = None
The queue on which events are placed.

Type Queue

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

Type int

auto_renew_fail = None
an optional function to be called if an exception occurs upon autorenewal. This will be called with the exception (or failure, when using *soco.events_twisted*) as its only parameter. This function must be threadsafe (unless *soco.events_twisted* is being used).

Type function

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.
- **is_autorenew** (*bool, optional*) – Whether this is an autorenewal.

unsubscribe()
Unsubscribe from the service's events. Once unsubscribed, a Subscription instance should not be reused

send_event(*event*)
Send an *Event* to self.callback or self.events. If self.callback is set and is callable, it will be called with the *Event* as the only parameter. Otherwise the *Event* will be sent to self.events. As self.callback is not threadsafe, it should be set only if *soco.events_twisted.Subscription* is being used.

Parameters **event** (Event) – The *Event* to send to self.callback or self.events.

time_left
The amount of time left until the subscription expires (seconds) If the subscription is unsubscribed (or not yet subscribed), *time_left* is 0.

Type int

class soco.events_base.**SubscriptionsMap**
Maintains a mapping of sids to *soco.events.Subscription* instances and the thread safe lock to go with it. Registers each subscription to be unsubscribed at exit.

SubscriptionsMapTwisted inherits from this class.

subscriptions = None
Thread safe mapping. Used to store a mapping of sid to subscription

Type weakref.WeakValueDictionary

subscriptions_lock = None
for use with *subscriptions*

Type threading.Lock

register(*subscription*)
Register a subscription by updating local mapping of sid to subscription and registering it to be unsubscribed at exit.

Parameters **subscription** (*soco.events.Subscription*) – the subscription to be registered.

unregister(*subscription*)
Unregister a subscription by updating local mapping of sid to subscription instances.

Parameters **subscription** (*soco.events.Subscription*) – the subscription to be unregistered.

When using *soco.events_twisted*, an instance of *soco.events_twisted.Subscription* will be unregistered.

get_subscription(*sid*)
Look up a subscription from a sid.

Args: sid(str): The sid from which to look up the subscription.

Returns: *soco.events.Subscription*: The subscription relating to that sid.

When using *soco.events_twisted*, an instance of *soco.events_twisted.Subscription* will be returned.

count
The number of active subscriptions.

Type int

1.10.2.10 soco.events_twisted module

Classes to handle Sonos UPnP Events and Subscriptions.

The *Subscription* class from this module will be used in *soco.services* if *config.EVENTS_MODULE* is set to point to this module.

Example

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

```
from __future__ import print_function
import logging
logging.basicConfig()
import soco
from pprint import pprint

from soco import events_twisted
soco.config.EVENTS_MODULE = events_twisted
from twisted.internet import reactor

def print_event(event):
    try:
        pprint (event.variables)
    except Exception as e:
        pprint ('There was an error in print_event:', e)

def main():
    # pick a device at random and use it to get
    # the group coordinator
    device = soco.discover().pop().group.coordinator
    print (device.player_name)
    sub = device.renderingControl.subscribe().subscription
    sub2 = device.avTransport.subscribe().subscription
    sub.callback = print_event
    sub2.callback = print_event

    def before_shutdown():
        sub.unsubscribe()
        sub2.unsubscribe()
        events_twisted.event_listener.stop()

    reactor.addSystemEventTrigger(
        'before', 'shutdown', before_shutdown)

if __name__=='__main__':
    reactor.callWhenRunning(main)
    reactor.run()
```

class soco.events_twisted.**Resource**
Fake Resource class to use when building docs

class soco.events_twisted.**EventNotifyHandler**
Handles HTTP NOTIFY Verbs sent to the listener server. Inherits from *soco.events_base.EventNotifyHandlerBase*.

render_NOTIFY(*request*)
Serve a NOTIFY request by calling *handle_notification* with the headers and content.

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

Runs an http server which is an endpoint for NOTIFY requests from Sonos devices. Inherits from *soco.events_base.EventListenerBase*.

port = None
set at *listen*

Type twisted.internet.tcp.Port

listen(*ip_address*)
Start the event listener listening on the local machine at port 1400 (default). If this port is unavailable, the listener will attempt to listen on the next available port, within a range of 100.

Make sure that your firewall allows connections to this port.

This method is called by *soco.events_base.EventListenerBase.start*

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

Parameters **ip_address** (*str*) – The local network interface on which the server should start listening.

Returns The port on which the server is listening.

Return type int

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

stop_listening(*address*)
Stop the listener.

class soco.events_twisted.**Subscription**(*service*, *event_queue=None*)
A class representing the subscription to a UPnP event. Inherits from *soco.events_base.SubscriptionBase*.

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.

callback = None
callback function to be called whenever an *Event* is received. If it is set and is callable, the callback function will be called with the *Event* as the only parameter and the Subscription's event queue won't be used.

Type function

subscribe(*requested_timeout=None*, *auto_renew=False*, *strict=True*)
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.

This method calls *events_base.SubscriptionBase.subscribe*.

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.

- **strict** (*bool, optional*) – If True and an Exception occurs during execution, the returned Deferred will fail with a Failure which will be passed to the applicable errback (if any has been set by the calling code) or, if False, the Failure will be logged and the Subscription instance will be passed to the applicable callback (if any has been set by the calling code). Default `True`.

Returns A Deferred the result of which will be the Subscription instance and the subscription property of which will point to the Subscription instance.

Return type Deferred

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.

This method calls *events_base.SubscriptionBase.renew*.

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.
- **is_autorenew** (*bool, optional*) – Whether this is an autorenewal. Default `False`.
- **strict** (*bool, optional*) – If True and an Exception occurs during execution, the returned Deferred will fail with a Failure which will be passed to the applicable errback (if any has been set by the calling code) or, if False, the Failure will be logged and the Subscription instance will be passed to the applicable callback (if any has been set by the calling code). Default `True`.

Returns A Deferred the result of which will be the Subscription instance and the subscription property of which will point to the Subscription instance.

Return type Deferred

unsubscribe()

Unsubscribe from the service's events. Once unsubscribed, a Subscription instance should not be reused

This method calls *events_base.SubscriptionBase.unsubscribe*.

Parameters **strict** (*bool, optional*) – If True and an Exception occurs during execution, the returned Deferred will fail with a Failure which will be passed to the applicable errback (if any has been set by the calling code) or, if False, the Failure will be logged and the Subscription instance will be passed to the applicable callback (if any has been set by the calling code). Default `True`.

Returns A Deferred the result of which will be the Subscription instance and the subscription property of which will point to the Subscription instance.

Return type Deferred

class soco.events_twisted.**SubscriptionsMapTwisted**

Maintains a mapping of sids to *soco.events_twisted.Subscription* instances. Registers each subscription to be unsubscribed at exit.

Inherits from *soco.events_base.SubscriptionsMap*.

register(*subscription*)

Register a subscription by updating local mapping of sid to subscription and registering it to be unsubscribed at exit.

Parameters **subscription** (*soco.events_twisted.Subscription*) – the subscription to be registered.

subscribing()
Called when the *Subscription.subscribe* method commences execution.

finished_subscribing()
Called when the *Subscription.subscribe* method completes execution.

count
The number of active or pending subscriptions.

Type int

1.10.2.11 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.

exception soco.exceptions.**SoCoNotVisibleException**
Raised when a command intended for a visible speaker is called on an invisible one.

exception soco.exceptions.**NotSupportedException**
Raised when something is not supported by the device

exception soco.exceptions.**EventParseException**(*tag*, *metadata*, *cause*)
Raised when a parsing exception occurs during event handling.

tag
The tag for which the exception occured

Type str

metadata
The metadata which failed to parse

Type str

__cause__
The original exception

Type Exception

Parameters

- **tag** (*str*) – The tag for which the exception occured
- **metadata** (*str*) – The metadata which failed to parse
- **cause** (*Exception*) – The original exception

class soco.exceptions.**SoCoFault**(*exception*)
Class to represent a failed object instantiation.

It rethrows the exception on common use.

exception
The exception which will be thrown on use

Parameters **exception** (*Exception*) – The exception which should be thrown on use

1.10.2.12 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={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={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.

Properties are available to get and set the group *volume* and the group *mute* state, and the *set_relative_volume()* method can be used to make relative adjustments to the group volume, e.g.:

```
>>> device.group.volume = 25
>>> device.group.volume
25
>>> device.group.set_relative_volume(-10)
15
>>> device.group.mute
>>> False
>>> device.group.mute = True
>>> device.group.mute
True
```

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
A description of the group.

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

Type str

short_label
A short description of the group.

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

Type str

volume
The volume of the group.

An integer between 0 and 100.

Type int

mute
The mute state for the group.

True or False.

Type bool

set_relative_volume(*relative_group_volume*)
Adjust the group volume up or down by a relative amount.

If the adjustment causes the volume to overshoot the maximum value of 100, the volume will be set to 100. If the adjustment causes the volume to undershoot the minimum value of 0, the volume will be set to 0.

Note that this method is an alternative to using addition and subtraction assignment operators (+=, -=) on the *volume* property of a *ZoneGroup* instance. These operators perform the same function as *set_relative_volume()* but require two network calls per operation instead of one.

Parameters **relative_group_volume** (*int*) – The relative volume adjustment. Can be positive or negative.

Returns The new group volume setting.

Return type int

Raises ValueError – If relative_group_volume cannot be cast as an integer.

1.10.2.13 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

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.

1.10.2.14 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.

build_album_art_full_uri(*url*)
Ensure an Album Art URI is an absolute URI.

Parameters **url** (*str*) – the album art URI.

Returns An absolute URI.

Return type str

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

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

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

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_information('artists', 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_information('artists',
                              subcategories=['Metallica'])
```

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

```
get_music_library_information('artists',
                              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_information('artists', start=0, max_items=100)
get_music_library_information('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_information('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 sub-categories 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
whether the music library is in the process of being updated.

Type bool

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
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.

Type str

list_library_shares()
Return a list of the music library shares.

Returns The music library shares, which are strings of the form `'//hostname_or_IP/share_path'`.

Return type list

delete_library_share(*share_name*)
Delete a music library share.

Parameters **share_name** (*str*) – the name of the share to be deleted, which should be of the form `'//hostname_or_IP/share_path'`.

Raises *SoCoUPnPException*

1.10.2.15 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**
A UPnP Action and its arguments.

Create new instance of ActionBase(name, in_args, out_args)

class soco.services.**Argument**
A UPnP Argument and its type.

Create new instance of ArgumentBase(name, vartype)

class soco.services.**Vartype**
An argument type with default value and range.

Create new instance of VartypeBase(datatype, default, list, range)

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
The *SoCo* instance to which UPnP Actions are sent

Type *SoCo*

service_type = None
The UPnP service type.

Type str

version = None
The UPnP service version.

Type str

base_url = None
The base URL for sending UPnP Actions.

Type str

control_url = None
The UPnP Control URL.

Type str

scpd_url = None
The service control protocol description URL.

Type str

event_subscription_url = None
The service eventing subscription URL.

Type str

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

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

compose_args(*action_name*, *in_argdict*)
Compose the argument list from an argument dictionary, with respect for default values.

Parameters

- **action_name** (*str*) – The name of the action to be performed.
- **in_argdict** (*dict*) – Arguments as a dict, e.g. `{'InstanceID': 0, 'Speed': 1}`. The values can be a string or something with a string representation.

Returns a list of `(name, value)` tuples.

Return type list

Raises

- `AttributeError` – If this service does not support the action.
- `ValueError` – If the argument lists do not match the action signature.

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*, ***kwargs*)
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, as an alternative to `kwargs`.
- **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.

- **kwargs** – Relevant arguments for the command.

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

Return type dict

Raises

- `AttributeError` – If this service does not support the action.
- `ValueError` – If the argument lists do not match the action signature.
- *SoCoUPnPException* – if a SOAP error occurs.
- *UnknownSoCoException* – if an unknown UPnP error occurs.
- `requests.exceptions.HTTPError` – if an http error occurs.

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*, *strict=True*)

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 `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.
- **strict** (*bool, optional*) – If True and an Exception occurs during execution, the Exception will be raised or, if False, the Exception will be logged and the Subscription instance will be returned. Default `True`.

Returns an instance of *Subscription*, representing the new subscription. If config.EVENTS_MODULE has been set to refer to `events_twisted`, a deferred will be returned with the Subscription as its result and deferred.subscription will be set to refer to the Subscription.

Return type *Subscription*

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

actions

The service's actions with their arguments.

Returns A list of Action namedtuples, consisting of action_name (str), in_args (list of Argument namedtuples, consisting of name and argtype), and out_args (ditto).

Return type list(*Action*)

The return value looks like this:

```
[
    Action(
        name='GetMute',
        in_args=[
            Argument(name='InstanceID', ...),
            Argument(
                name='Channel',
                vartype='string',
                list=['Master', 'LF', 'RF', 'SpeakerOnly'],
                range=None
            )
        ],
        out_args=[
            Argument(name='CurrentMute, ...)
        ]
    )
    Action(...)
]
```

Its string representation will look like this:

```
GetMute(InstanceID: ui4, Channel: [Master, LF, RF, SpeakerOnly])

-> {CurrentMute: boolean}
```

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=<Vartype>),
        Argument(name='DesiredDateFormat', vartype=<Vartype>)],
    out_args=[]
)
```

event_vars
The service's eventable variables.

Returns A list of (variable name, data type) tuples.

Return type list(tuple)

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.

Parameters

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

class soco.services.**AudioIn**(*soco*)
Sonos audio in service, for functions related to RCA audio input.

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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.

1.10.2.16 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 or an announcement and then back again to what was playing previously.

Warning: Sonos has introduced control via Amazon Alexa. A new cloud queue is created and at present there appears no way to restart this queue from snapshot. Currently if a cloud queue was playing it will not restart.

Warning: This class is designed to be created used and destroyed. It is not designed to be reused or long lived. The init sets up defaults for one use.

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.

1.10.2.17 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*) – The 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.

1.10.2.18 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*, *alternative_not_referable=False*)
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.
- **alternative_not_referable** (*bool*) – (optional) Indicate that alternative cannot be used as a sphinx reference

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.utils.**first_cap**(*string*)
Return upper cased first character

1.10.2.19 soco.xml module

This class contains XML related utility functions.

soco.xml.**NAMESPACES = {'': 'urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/', 'dc': 'http://**
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'
```

1.11 SoCo releases

1.11.1 SoCo 0.21 release notes

SoCo 0.21 (Released on 2021-01-17), contains significant improvements to the speaker discovery process, which should address most of the common problems that are encountered.

Multi-household Sonos systems can now be discovered and controlled.

The *AudioIn* service has been added, providing access to Line In operations and events.

A number of additional miscellaneous speaker state controls have been provided, for speaker buttons, fixed volume output, and Trueplay. The battery status of Sonos Move speakers can also now be obtained. The *music_source* property on SoCo objects provides a convenient way to determine what type of source is being played by a speaker.

SoCo 0.21 is now fully Python 3.9 compatible, requires Python 3.5+, and is no longer compatible with Python 2.x.

1.11.1.1 New Features and Improvements

- Add support for **network scan discovery**, including allowing multi-household systems to be discovered: pull requests #733, #755, and #770.
- Add the **AudioIn service** for access to Line In operations and events: PR #777.
- Add the ability to inspect and set **Fixed Volume** output: PR #773.
- Add the ability to inspect and set whether **speaker buttons** are enabled: PR #774.
- Add the ability to inspect and set **Trueplay** enablement: PR #775.
- Add the ability to determine the **battery state** of Sonos Move speakers: PR #756.

1.11.1.2 Bug Fixes

- Improve zone group state caching to accommodate multi-household systems: PR #656. (Note: possible **breaking change**: if you've previously been setting the `soco.core.zone_group_state_shared_cache.enabled` property, this property is no longer global but is instead now a private property of SoCo instances.)
- When restoring snapshots, do not try to restore bass/treble/loudness on devices with fixed volume enabled: PR #772.
- Ensure that all relevant NICs and IP addresses are included in multicast speaker discovery: PR #767.

1.11.1.3 Developer Improvements

- Numerous fixes to allow the documentation to build cleanly: PR #753.
- Full Python 3.9 compatibility, including updated check jobs: PRs #745 and #751.
- Code changes to allow Pylint and Black to be updated to their most recent versions: PRs #748 and #749.

1.11.1.4 List of Changes Associated with the 0.21 Milestone

See: https://github.com/SoCo/SoCo/milestone/17?closed=1

1.11.2 SoCo 0.20 release notes

SoCo 0.20 is the latest increment to the SoCo module. Among the additions this time are support for adding stereo pairs, proper categorization of Sonos Amp as a playbar to add proper support for 'night sound' and 'speech enhancement' and finally a fix for a long running issue where vendor extended DIDL-Lite classes would cause events to crash without specific code added for each one. See the full list of additions and bugfixes below.

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

1.11.2.1 New Features and Improvements

- Add support for creating and separating stereo pairs of speakers. Note: works with dissimilar Sonos speakers if required. Pull request #704.
- Add support for autogenerating vendor extended DIDL-Lite classes. Pull request #713. This should fix all the problems where SoCo would crash if some vendor specific data type is unknown.
- Categorize Sonos Amp as a playbar in order to provide support for 'night sound' and 'speech enhancement'. Pull request #721
- If port 1400 is in use, the next available 100 ports will be tried. Pull request #724.

1.11.2.2 Bugfixes

- Fix bug where data_structures_upgrade would fail on items that has no uri. Issue #702.
- Process share browsing correctly. Issue #717. Credit to @Sonosy for the fix.

1.11.2.3 Developer improvements

- Format all soco main, test and example code with the black code formatter (https://github.com/psf/black) and make it mandatory going forward including a TravisCI check. Pull request #706.
- Improve `test_remove_playlist_bad_id()` to handle the case of no existing playlists. Pull request #726, fixes issue #725.

1.11.3 SoCo 0.19 release notes

SoCo 0.19 is the latest increment to the SoCo module. Among the additions this time are added methods for library share handling, new methods for relative and group volume handling and a new DIDL-Lite class used for certain podcasts. See the full list of additions and bugfixes below.

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

1.11.3.1 New Features and Improvements

- Added class *DidlRecentShow* to the *data_structures* module to implement the added `object.item.audioItem.musicTrack.recentShow` DIDL-Lite object type. Used for podcasts etc. Pull request #677.
- Add support for Python 3.8, pull reques #679
- Add methods *list_library_shares()* and *delete_library_share()* to *MusicLibrary*. Partially addresses issue #678.
- Add a *balance* property to the *SoCo* class, allowing get/set of speaker balance, pull request #693. Addresses issue #692. Credit to @tephlon for the idea and the majority of the implementation.
- Add the *set_relative_volume()* method to the *SoCo* class, pull request #687
- Add unit test for *soco.music_library.MusicLibrary.delete_library_share()* method, pull request #694
- Add deprecation warning concerning the removal of Python 2.7 support, pull request #697
- Add group volume operations, pull request #688

1.11.3.2 Bugfixes

- Fixed broken link in loudness docstring, issue #671
- In *soco.events*, fixed bug affecting some users in code to determine system's own IP address. Some systems requires a valid port to be used (not port 0), so we use `config.EVENT_LISTENER_PORT`. Pull request #680.
- Copy metadata from *DidlItem* to *MusicServiceItem* in *get_queue()* and events. Pull request #589. Closes issues #535, #547 and #552.
- Fixed a bug (avoid trying to iterate a None) in the *discovery* module, commit c8e4a24

1.11.4 SoCo 0.18 release notes

SoCo 0.18 adds lots of small improvements to the events functionality plus a major addition in the form of allowing choice of how the event listener is implemented. Besides that there is a logging improvement. Details are below.

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

1.11.4.1 New Features and Improvements

- Allow the user to choose how the event listener is implemented and a lot of other event code improvements as outlined below. (Pull request #602).

  - A major feature addition is to allow the user to choose how the event listener is implemented. The default is for the event listener to use the requests library and run in a thread. This update allows the user to run the event listener using the twisted.internet library, by setting the `config.EVENTS_MODULE` module to point to the *soco.events_twisted* module. See the example in events_twisted.
  - Stops the event listener when the last active subscription is unsubscribed.
  - Raise *soco.exceptions.SoCoException* on an attempt to subscribe a subscription more than once (use *soco.events.Subscription.renew()* instead).
  - Allow an optional strict parameter for *soco.events.Subscription.subscribe()*, *soco.events.Subscription.renew()* and *soco.events.Subscription.unsubscribe()*. If set to False, Exceptions will be logged rather than raised. Default: True
  - Upon autorenewal, call *soco.events.Subscription.renew()* with the strict flag set to False, so that any Exception is logged, not raised. This is because there is no calling code to catch an Exception.
  - Provide for calling code to set `soco.events.Subscription.auto_renew_fail` to refer to a callback function. If an Exception occurs upon autorenewal, this callback will be called with the Exception as the sole parameter.
  - If an Exception occurs upon `subscribe()` or `renew()`, cancel the subscription, unless the Exception was a SoCoException on `subscribe()`. For example, if an Exception occurs because the network went down, the subscription will be canceled.
  - Use a threading lock with `subscribe()`, `renew()` and `unsubscribe()`, because autorenewal occurs from a thread.

- Add a simple *soco.data_structures.DidlPlaylistContainerTracklist* class to the *soco.data_structures* module (Pull request #645). The class is used by Sonos when Sonos Speakers are controlled by Spotify Connect. The absence of the class from the data_structures module causes errors. This fixes the error message reported in pull request #639.
- Remove logging of UPnP failures (Pull request #640

1.11.5 SoCo 0.17 release notes

SoCo 0.17 adds a single new feature and updates SoCo to work on top of the API changes that Sonos introduced with the 10.1 software update.

> **Warning:** The changes to SoCo to accommodate the Sonos API changes as of version 10.1 are **backwards incompatible**. This means that if SoCo is updated to version 0.17, then *it will be necessary to update the Sonos software to 10.1* at the same time.

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

1.11.5.1 New Features and Improvements

- Add the `is_soundbar` property to the SoCo class to indicate whether or not the current instance represents a Play:Bar, a Play:Base, or a Beam and, when appropriate, enable features like Night and Dialog mode. (Pull request #637). (Fixes #633).

1.11.5.2 Bugfixes

- Fix discovery which was broken as a consequence of API changes in Sonos software version 10.1. (Commit f532cad)
- Fix parsing of favorites which was broken as a consequence of API changes in Sonos software version 10.1. (Commit 58efcb6)

1.11.6 SoCo 0.16 release notes

SoCo 0.16 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.

1.11.6.1 New Features and Improvements

- Allow the user to configure the event listener IP address that is sent to the Sonos speakers. The default is to auto detect, but it can now be overridden. This allows for more complex network configurations (e.g. using Docker containers) to be supported. (#604)
- The `play_uri` method now accepts title arguments that need XML escaping. (#605)
- A harmless "Could not handle track info" warning has been removed. (#606)
- Let `from_didl_string` throw `DIDLMetadataErrors`, allowing them to be caught in the event handling code. (#601)
- Added support for `object.item.audioItem.audioBook` (#618)

1.11.6.2 Bugfixes

- Fix `DidlMusicAlbum` inheriting fields from `DidlAudioItem` instead of `DidlAlbum` (#592)

1.11.7 SoCo 0.15 release notes

SoCo 0.15 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.

1.11.7.1 New Features and Improvements

- Add `__enter__` and `__exit__` methods to Subscription, for automatic unsubscription in a with-block (#563)
- Add `__enter__` and `__exit__` methods to Snapshot, for automatic snapshot and restore in a with block (#588)
- Handle default value / allowed value range in `Service.iter_actions` and format the resulting actions (#573)
- Allow keyword arguments in Service commands (#573)
- Auto deploy new tagged releases to PyPI (#593)
- Documentation updates (#580)

1.11.7.2 Bugfixes

- Prevent parsing exceptions during event handling from killing the exception thread. Instead, return a `DidlFault`, which will reraise the exception when the user tries to use it (#567)
- Fixed the set returned by `discover()` being modified later (#582)
- Fixed regression in `send_command` (#577)
- Fixed regression due to removed deprecated methods (#596)
- Improved error handling with speakers not associated to a room (#555)

1.11.7.3 Backwards Compatability

- Dropped support for Python 3.3 #527 (#527)
- Removed the deprecated methods which were moved in 0.12 from `core.py` to `music_library.py` and move the assoctiated tests (#542)

1.11.8 SoCo 0.14 release notes

SoCo 0.14 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.

1.11.8.1 New Features and Improvements

- Add support for Sonos favorites, which can now be browsed and played through the usual methods. (#478)
- Revised the play_local_files examples including a off-by-one bug fix, configuration as command line argument, IP address auto detection and more robust Sonos player selection. (#570)
- Allow keyword arguments in Service commands (#573)
- Handle QueueID properly in event xml. (#546)
- Further documentation updates (#540, #569)

1.11.8.2 Bugfixes

- Small bugfix to stop an error where `None` would be returned by `metadata.findtext`. Instead, an empty string is returned. (#539)
- Fix a race that could lead to events being missed shortly after a subscription was started. (#533)
- Don't throw exceptions when parsing metadata with missing/empty tags, to fix event errors. (#467)

1.11.9 SoCo 0.13 release notes

SoCo 0.13 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.

1.11.9.1 New Features and Improvements

- The IP address used by the events listener can be configured (#444)
- Add support for night mode (#421) and dialog mode (#422) on devices supporting the respective feature.
- Add queue-able data structures for the music service items (#455)
- Add a method for queueing multiple items with a single request (#470)
- Add methods to get and set the uri(s) of a `DidlObject`. (#482)
- Add support for line in from other speakers (#460)
- Enhance `add_to_queue()` to with optional position argument (#471)
- Added `by_name` function to discovery to be able to get a device by `player_name` (#487)
- allow choice of how to play streams in `play_uri` (override Sonos default with `force_radio=True`) (#491)
- Added `ramp_to_volume()` method to smoothly change the volume (#506)
- Added FAQ, documentation and two examples to explain using SoCo's Snapshot function (#493)
- Update documentation for `add_uri_to_queue` (#503)
- Added a FAQ section to the docs with play_uri and play local files answers (#481)
- A few queue related micro examples was added to the examples page in the docs (#484)
- Further documentation updates (#435, #436, #459, #476, #489, #496, #522)

1.11.9.2 Bugfixes

- Fixes an issue where restarting an application that had subscribed to events sometimes causes an error when the events are delivered to the new instance (#437)
- Fixes an issue where multiple threads trying to subscribe to events in parallel would sometimes cause SoCo to attempt to create multiple event listener servers and fail on `socket.bind()`. (#437)
- Fixes an issue where SoCo would not recognize `object.container.playlistContainer.sonos-favorite` when receiving events (#438).
- Fixes a bug in `play_uri` where it would not play a http or https prefixed radio stream due to a change in the Sonos API (issue #434). This change fixes it by replacing the two http type prefixes with Sonos' `x-rincon-mp3radio://` prefix (#434, #443)
- Fixes an exception being raised on Windows with `discover`. The error was caused by `socket.getsockname` raising an exception on Windows with and unconnected unbound socket. Fixed by now simply logging the sockets. (#445)
- Change to use SoCo method to determine coordinator in Snapshot (#529, #519)
- Prevent error when queue started from Alexa and using snapshot. Currently there is no was to restart a cloud queue from SoCo, this PR just prevents Snapshot causing an error. (#530, #521)
- Fix `add_multiple_to_queue` fails with too many items (#488)
- Fixed log level (#534)

1.11.9.3 Backwards Compatability

- Dropped support for Python 2.6 (#325, #526) and added support for 3.6 (#528)

1.11.10 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.

1.11.10.1 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)

1.11.10.2 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)

1.11.10.3 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)

1.11.11 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.

1.11.11.1 Bugfixes

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

1.11.12 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.

1.11.12.1 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)

1.11.12.2 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)

1.11.12.3 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)

1.11.13 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.

1.11.13.1 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)

1.11.13.2 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)

1.11.13.3 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)

1.11.13.4 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.

1.11.14 SoCo 0.9 release notes

1.11.14.1 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>])
```

(continues on next page)

(continued from previous page)

```
>>> # 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)

1.11.14.2 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`.

1.11.14.3 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.

1.11.15 SoCo 0.8 release notes

1.11.15.1 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 *soco.services.Service.subscribe* method, which returns a *soco.events.Subscription* object. To unsubscribe, call the *soco.events.Subscription.unsubscribe* method on the returned object. (#121, #130)
- Support for reading and setting crossfade (#165)

1.11.15.2 Improvements

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

1.11.15.3 Backwards Compatability

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

1.11.16 SoCo 0.7 release notes

1.11.16.1 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).

1.11.16.2 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")
```

1.11.17 SoCo 0.6 release notes

1.11.17.1 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.

1.11.17.2 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.

1.11.17.3 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.

1.12 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.

1.12.1 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
```

1.12.2 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 .
```

1.12.3 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
```

1.12.4 Unit test code structure and naming conventions

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

1.12.4.1 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`.

1.12.4.2 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`

1.12.5 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).

1.12.5.1 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.

1.12.6 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.

1.12.6.1 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.

1.13 Release Procedures

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

1.13.1 Preparations

- Verify the version number stated in the release ticket (according to semantic versioning. Tag names should be prefixed with `v`.
- Create the release notes RST document in `doc/releases` by copying contents from the release notes issue. Texts can be rewritten for legibility.
- Verify that all tests pass locally and on all supported versions of Python via Travis-CI (the status is visible on the project frontpage on GitHub).

1.13.2 Create and Publish

- Update the version number in `__init__.py` (see example) and commit.
- (**If any changes other than the version number was made** in preparation for the release, push the release commit to GitHub before proceeding, to ensure that all the continuous integration passes. The automatic deployment to PyPI mentioned below, will not work if continuous integration fails.)

- 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, and will automatically deploy the new version to PyPI (see #593)

```
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.

1.13.3 Wrap-Up

- Close the milestone and issues for the release.
- Update the version number in `__init__.py` with an added "+" to indicate development status (see example).
- Share the news!

1.13.4 Preparation for next release

- Define the next version number and expected release date (3 month after the current release date, as per #524)).
- Create the milestone and set the release date.
- Create an issue for the upcoming release (tagged as Release), and one for the corresponding release notes.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

T

U

V

W

Z