pyspotify

Release 2.1.1

Nov 17, 2019

Contents

pyspotify provides a Python interface to Spotify's online music streaming service.

With pyspotify you can access music metadata, search in Spotify's library of 20+ million tracks, manage your Spotify playlists, and play music from Spotify. All from your own Python applications.

pyspotify uses CFFI to make a pure Python wrapper around the official libspotify library. It works on CPython 2.7 and 3.5+, as well as PyPy 2.7 and 3.5+. It is known to work on Linux and macOS. Windows support should be possible, but is awaiting a contributor with the interest and knowledge to maintain it.

CHAPTER 1

libspotify's deprecation

Note that as of May 2015 libspotify is officially deprecated by Spotify and is no longer actively maintained.

Spotify has published newer libraries intended for Android and iOS development, as well as web APIs to access track metadata and manage playlists. Though, for making apps with Spotify playback capabilities, on any other platform than Android and iOS, there is currently no alternative to libspotify.

libspotify has been the main way of integrating with Spotify since 2009, and is today a part of numerous open source projects and commercial applications, including many receivers and even cars. There's no guarantees, but one can hope that the large deployment of libspotify means that the library will continue to work with the Spotify service for a long time into the future.

CHAPTER 2

Project resources

- Documentation
- Source code
- Issue tracker

CHAPTER 3

User's guide

3.1 Installation

pyspotify is packaged for various operating systems and in multiple Linux distributions. What way to install pyspotify is best for you depends upon your OS and/or distribution.

3.1.1 Debian/Ubuntu: Install from apt.mopidy.com

The Mopidy project runs its own APT archive which includes pyspotify built for:

- Debian stretch (oldstable), which also works for Ubuntu 16.04 LTS.
- Debian buster (stable), which also works for Ubuntu 18.04 LTS and newer.

The packages are available for multiple CPU architectures: i386, amd64, armel, and armhf (compatible with Raspbian and all Raspberry Pi models).

To install and receive future updates:

1. Add the archive's GPG key:

```
wget -q -O - https://apt.mopidy.com/mopidy.gpg | sudo apt-key add -
```

2. If you run Debian stretch or Ubuntu 16.04 LTS:

```
sudo wget -q -O /etc/apt/sources.list.d/mopidy.list https://apt.mopidy.com/
↪stretch.list
```

   Or, if you run any newer Debian/Ubuntu distro:

```
sudo wget -q -O /etc/apt/sources.list.d/mopidy.list https://apt.mopidy.com/buster.
↪list
```

3. Install pyspotify and all dependencies:

```
sudo apt-get update
sudo apt-get install python-spotify
```

3.1.2 Arch Linux: Install from AUR

If you are running Arch Linux on x86 or x86_64, you can install pyspotify using the python2-pyspotify package found in AUR.

1. To install pyspotify with all dependencies, run:

```
yay -S python2-pyspotify
```

Note: AUR does not provide libspotify for all CPU architectures e.g. arm. See *installing from source* in these cases.

3.1.3 OS X: Install wheel package from PyPI with pip

From PyPI, you can install precompiled wheel packages of pyspotify that bundle libspotify. The packages should work on all combinations of:

- OS X 10.6 and newer
- 32-bit and 64-bit
- Apple-Python, Python.org-Python, Homebrew-Python

1. Make sure you have a recent version of pip, which will default to installing a wheel package if available:

```
pip install --upgrade pip
```

2. Install pyspotify:

```
pip install pyspotify
```

3.1.4 OS X: Install from Homebrew

The Mopidy project maintains its own Homebrew tap which includes pyspotify and its dependencies.

1. Install Homebrew.
2. Make sure your installation is up to date:

```
brew update
brew upgrade --all
```

3. Install pyspotify from the mopidy/mopidy tap:

```
brew install mopidy/mopidy/python-spotify
```

3.1.5 Install from source

If you are on Linux, but your distro don't package pyspotify, you can install pyspotify from PyPI using the pip installer. However, since pyspotify is a Python wrapper around the libspotify library, pyspotify necessarily depends on libspotify and it must be installed first.

libspotify

libspotify is provided as a binary download for a selection of operating systems and CPU architectures from our unofficial libspotify archive. If libspotify isn't available for your OS or architecture, then you're out of luck and can't use pyspotify either.

To install libspotify, use one of the options below, or follow the instructions in the README file of the libspotify tarball.

Debian/Ubuntu

If you're running a Debian-based Linux distribution, like Ubuntu, you can get Debian packages of libspotify from apt.mopidy.com. Follow the instructions *above* to make the apt.mopidy.com archive available on your system, then install libspotify:

```
sudo apt-get install libspotify-dev
```

Arch Linux

libspotify for x86 and x86_64 is packaged in AUR. To install libspotify, run:

```
yay -S libspotify
```

Note: AUR only provides libspotify binaries for x86 and x86_64 CPUs. If you require libspotify for a different CPU architecture you'll need to download it from our unofficial libspotify archive instead.

OS X

If you're using Homebrew, it has a formula for libspotify in the homebrew/binary tap:

```
brew install homebrew/binary/libspotify
```

Warning: There's an issue with building pyspotify against libspotify on OS X where the pyspotify installation fails with "Reason: image not found".

A known workaround is to create a symlink after installing libspotify, but before installing pyspotify:

```
ln -s /usr/local/opt/libspotify/lib/libspotify.12.1.51.dylib \
/usr/local/opt/libspotify/lib/libspotify
```

Alternatively, the mopidy/mopidy Homebrew tap has a libspotify formula which includes the workaround:

```
brew install mopidy/mopidy/libspotify
```

For details, or if you have a proper fix for this, see #130.

Build tools

To build pyspotify, you need a C compiler, Python development headers, and libffi development headers. All of this is easily installed using your system's package manager.

Debian/Ubuntu

If you're on a Debian-based system, you can install the pyspotify build dependencies by running:

```
sudo apt-get install build-essential python-dev python3-dev libffi-dev
```

Arch Linux

If you're on Arch Linux, you can install the pyspotify build dependencies by running:

```
sudo pacman -S base-devel python2 python
```

OS X

If you're on OS X, you'll need to install the Xcode command line developer tools. Even if you've already installed Xcode from the App Store, e.g. to get Homebrew working, you should run this command:

```
xcode-select --install
```

Note: If you get an error about `ffi.h` not being found when installing the cffi Python package, try running the above command.

pyspotify

With libspotify and the build tools in place, you can finally build pyspotify.

To download and build pyspotify from PyPI, run:

```
pip install pyspotify
```

Or, if you have a checkout of the pyspotify git repo, run:

```
pip install -e path/to/my/pyspotify/git/clone
```

Once you have pyspotify installed, you should head over to *Quickstart* for a short introduction to pyspotify.

3.2 Quickstart

This guide will quickly introduce you to some of the core features of pyspotify. It assumes that you've already installed pyspotify. If you do not, check out *Installation*. For a complete reference of what pyspotify provides, refer to the *API reference*.

3.2.1 Application keys

Every app that use libspotify needs its own libspotify application key. Application keys can be obtained automatically and free of charge from Spotify.

1. Go to the Spotify developer pages and login using your Spotify account.
2. Find the libspotify application keys management page and request an application key for your application.
3. Once the key is issued, download the "binary" version. The "C code" version of the key will not work with pyspotify.
4. If you place the application key in the same directory as your application's main Python file, pyspotify will automatically find it and use it. If you want to keep the application key in another location, you'll need to set `application_key` in your session config or call `load_application_key_file()` to load the session key file correctly.

3.2.2 Creating a session

Once pyspotify is installed and the application key is in place, we can start writing some Python code. Almost everything in pyspotify requires a `Session`, so we'll start with creating a session with the default config:

```
>>> import spotify
>>> session = spotify.Session()
```

All config must be done before the session is created. Thus, if you need to change any config to something else than the default, you must create a `Config` object first, and then pass it to the session:

```
>>> import spotify
>>> config = spotify.Config()
>>> config.user_agent = 'My awesome Spotify client'
>>> config.tracefile = b'/tmp/libspotify-trace.log'
>>> session = spotify.Session(config)
```

3.2.3 Text encoding

libspotify encodes all text as UTF-8. pyspotify converts the UTF-8 bytestrings to Unicode strings before returning them to you, so you don't have to be worried about text encoding.

Similarly, pyspotify will convert any string you give it from Unicode to UTF-8 encoded bytestrings before passing them on to libspotify. The only exception is file system paths, like `tracefile` above, which is passed directly to libspotify. This is in case you have a file system which doesn't use UTF-8 encoding for file names.

3.2.4 Login and event processing

With a session we can do a few things, like creating objects from Spotify URIs:

```
>>> import spotify
>>> session = spotify.Session()
>>> album = session.get_album('spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album
Album(u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album.link
Link(u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album.link.uri
u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE'
```

But that's mostly how far you get with a fresh session. To do more, you need to login to the Spotify service using a Spotify account with the Premium subscription.

Warning: pyspotify and all other libspotify applications required a Spotify Premium subscription.

The Free Spotify subscription, or the old Unlimited subscription, will not work with pyspotify or any other applications using libspotify.

```
>>> import spotify
>>> session = spotify.Session()
>>> session.login('alice', 's3cretpassword')
```

For alternative ways to login, refer to the *login()* documentation.

The *login()* method is asynchronous, so we must ask the session to *process_events()* until the login has succeeded or failed:

```
>>> session.connection.state
<ConnectionState.LOGGED_OUT: 0>
>>> session.process_events()
>>> session.connection.state
<ConnectionState.OFFLINE: 1>
>>> session.process_events()
>>> session.connection.state
<ConnectionState.LOGGED_IN: 1>
```

Note: The connection state is a representation of both your authentication state and your offline mode. If libspotify has cached your user object from a previous session, it may authenticate you without a connection to Spotify's servers. Thus, you may very well be logged in, but still offline.

The connection state in the above example goes from the `LOGGED_OUT` state, to `OFFLINE`, to `LOGGED_IN`. If libspotify hasn't cached any information about your Spotify user account, the connection state will probably go directly from `LOGGED_OUT` to `LOGGED_IN`. Your application should be prepared for this.

For more details, see the *session.connection.state* documentation.

We only called *process_events()* twice, which may not be enough to get to the `LOGGED_IN` connection state. A more robust solution is to call it repeatedly until the *CONNECTION_STATE_UPDATED* event is emitted on the *Session* object and *session.connection.state* is `LOGGED_IN`:

```
>>> import threading
>>> logged_in_event = threading.Event()
>>> def connection_state_listener(session):
...     if session.connection.state is spotify.ConnectionState.LOGGED_IN:
```

(continues on next page)

(continued from previous page)

```
...         logged_in_event.set()
...
>>> session = spotify.Session()
>>> session.on(
...     spotify.SessionEvent.CONNECTION_STATE_UPDATED,
...     connection_state_listener)
...
>>> session.login('alice', 's3cretpassword')
>>> session.connection.state
<ConnectionState.LOGGED_OUT: 0>
>>> while not logged_in_event.wait(0.1):
...     session.process_events()
...
>>> session.connection.state
<ConnectionState.LOGGED_IN: 1>
>>> session.user
User(u'spotify:user:alice')
```

This solution works properly, but is a bit tedious. pyspotify provides an *EventLoop* helper thread that can make the *process_events()* calls in the background. With it running, we can simplify the login process:

```
>>> import threading
>>> logged_in_event = threading.Event()
>>> def connection_state_listener(session):
...     if session.connection.state is spotify.ConnectionState.LOGGED_IN:
...         logged_in_event.set()
...
>>> session = spotify.Session()
>>> loop = spotify.EventLoop(session)
>>> loop.start()
>>> session.on(
...     spotify.SessionEvent.CONNECTION_STATE_UPDATED,
...     connection_state_listener)
...
>>> session.connection.state
<ConnectionState.LOGGED_OUT: 0>
>>> session.login('alice', 's3cretpassword')
>>> session.connection.state
<ConnectionState.OFFLINE: 4>
>>> logged_in_event.wait()
>>> session.connection.state
<ConnectionState.LOGGED_IN: 1>
>>> session.user
User(u'spotify:user:alice')
```

Note that when using *EventLoop*, your event listener functions are called from the *EventLoop* thread, and not from your main thread. You may need to add synchronization primitives to protect your application code from threading issues.

3.2.5 Logging

pyspotify uses Python's standard `logging` module for logging. All log records emitted by pyspotify are issued to the logger named `spotify`, or a sublogger of it.

Out of the box, pyspotify is set up with `logging.NullHandler` as the only log record handler. This is the recommended approach for logging in libraries, so that the application developer using the library will have full

control over how the log records from the library will be exposed to the application's users. In other words, if you want to see the log records from pyspotify anywhere, you need to add a useful handler to the root logger or the logger named `spotify` to get any log output from pyspotify. The defaults provided by `logging.basicConfig()` is enough to get debug log statements out of pyspotify:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

If your application is already using `logging`, and you want debug log output from your own application, but not from pyspotify, you can ignore debug log messages from pyspotify by increasing the threshold on the "spotify" logger to "info" level or higher:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('spotify').setLevel(logging.INFO)
```

For more details on how to use `logging`, please refer to the Python standard library documentation.

If we turn on logging, the login process is a bit more informative:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import spotify
>>> session = spotify.Session()
>>> session.login('alice', 's3cretpassword')
DEBUG:spotify.session:Notify main thread
DEBUG:spotify.session:Log message from Spotify: 19:15:54.829 I [ap:1752] Connecting␣
↪to AP ap.spotify.com:4070
DEBUG:spotify.session:Log message from Spotify: 19:15:54.862 I [ap:1226] Connected to␣
↪AP: 78.31.12.11:4070
>>> session.process_events()
DEBUG:spotify.session:Notify main thread
DEBUG:spotify.session:Log message from Spotify: 19:17:27.972 E [session:926] Not all␣
↪tracks cached
INFO:spotify.session:Logged in
DEBUG:spotify.session:Credentials blob updated: 'NfFEO...'
DEBUG:spotify.session:Connection state updated
43
>>> session.user
User(u'spotify:user:alice')
```

3.2.6 Browsing metadata

When we're logged in, the objects we created from Spotify URIs becomes a lot more interesting:

```
>>> album = session.get_album('spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
```

If the object isn't loaded, you can call *`load()`* to block until the object is loaded with data:

```
>>> album.is_loaded
False
>>> album.name is None
True
>>> album.load()
Album('spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album.name
```

(continues on next page)

(continued from previous page)

```
u'Reach For Glory'
>>> album.artist
Artist(u'spotify:artist:4kjWnaLfIRcLJ1Dy4Wr6tY')
>>> album.artist.load().name
u'Blackmill'
```

The *Album* object give you the most basic information about an album. For more metadata, you can call *browse()* to get an *AlbumBrowser*:

```
>>> browser = album.browse()
```

The browser also needs to load data, but once its loaded, most related objects are in place with data as well:

```
>>> browser.load()
AlbumBrowser(u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> browser.copyrights
[u'2011 Blackmill']
>>> browser.tracks
[Track(u'spotify:track:4FXj4ZKMO2dSkqiAhV7L8t'),
 Track(u'spotify:track:1sYClIlZZsL6dVMVTxCYRm'),
 Track(u'spotify:track:1uY4O332HuqLIcSSJlg4NX'),
 Track(u'spotify:track:58qbTrCRGyjF9tnjvHDqAD'),
 Track(u'spotify:track:3RZzg8yZs5HaRjQiDiBIsV'),
 Track(u'spotify:track:4jIzCryeLdBgE671gdQ6QD'),
 Track(u'spotify:track:4JNpKcFjVFYIzt1D95dmi0'),
 Track(u'spotify:track:7wAtUSgh6wN5ZmuPRRXHyL'),
 Track(u'spotify:track:7HYOVVLd5XnfY4yyV5Neke'),
 Track(u'spotify:track:2YfVXi6dTux0x8KkWeZdd3'),
 Track(u'spotify:track:6HPKugiH3p0pUJBNgUQoou')]
>>> [(t.index, t.name, t.duration // 1000) for t in browser.tracks]
[(1, u'Evil Beauty', 228),
 (2, u'City Lights', 299),
 (3, u'A Reach For Glory', 254),
 (4, u'Relentless', 194),
 (5, u'In The Night Of Wilderness', 327),
 (6, u"Journey's End", 296),
 (7, u'Oh Miah', 333),
 (8, u'Flesh and Bones', 276),
 (9, u'Sacred River', 266),
 (10, u'Rain', 359),
 (11, u'As Time Goes By', 97)]
```

3.2.7 Downloading cover art

While we're at it, let's do something a bit more impressive; getting cover art:

```
>>> cover = album.cover(spotify.ImageSize.LARGE)
>>> cover.load()
Image(u'spotify:image:16eaba4959d5d97e8c0ca04289e0b1baaefae55f')
```

Currently, all covers are in JPEG format:

```
>>> cover.format
<ImageFormat.JPEG: 0>
```

The *Image* object gives access to the raw JPEG data:

```
>>> len(cover.data)
37204
>>> cover.data[:20]
'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00H\x00H\x00\x00'
```

For convenience, it also provides the same data encoded as a `data:` URI for easy embedding into HTML documents:

```
>>> len(cover.data_uri)
49631
>>> cover.data_uri[:60]
u''
```

If you're following along, you can try writing the image data out to files and inspect the result yourself:

```
>>> open('/tmp/cover.jpg', 'w+').write(cover.data)
>>> open('/tmp/cover.html', 'w+').write('<img src="%s">' % cover.data_uri)
```

3.2.8 Searching

Warning: The search API was broken at 2016-02-03 by a server-side change made by Spotify. The functionality was never restored.

Please use the Spotify Web API to perform searches.

If you don't have the URI to a Spotify object, another way to get started is to *search()*:

```
>>> search = session.search('massive attack')
>>> search.load()
Search(u'spotify:search:massive+attack')
```

A search returns lists of matching artists, albums, tracks, and playlists:

```
>>> (search.artist_total, search.album_total, search.track_total, track.playlist_
↪total)
(5, 50, 564, 125)
>>> search.artists[0].load().name
u'Massive Attack'
>>> [a.load().name for a in search.artists[:3]]
[u'Massive Attack',
 u'Kwanzaa Posse feat. Massive Attack',
 u'Massive Attack Vs. Mad Professor']
```

Only the first 20 items in each list are returned by default:

```
>>> len(search.artists)
5
>>> len(search.tracks)
20
```

The *Search* object can help you with getting *more()* results from the same query:

```
>>> search2 = search.more().load()
>>> len(search2.artists)
```

(continues on next page)

(continued from previous page)

```
0
>>> len(search2.tracks)
20
>>> search.track_offset
0
>>> search.tracks[0]
Track(u'spotify:track:67Hna13dNDkZvBpTXRIaOJ')
>>> search2.track_offset
20
>>> search2.tracks[0]
Track(u'spotify:track:3kKVqFF4pv4EXeQe428zl2')
```

You can also do searches where Spotify tries to figure out what you mean based on popularity, etc. instead of exact token matches:

```
>>> search = session.search('mas').load()
Search(u'spotify:search:mas')
>>> search.artists[0].load().name
u'X-Mas Allstars'

>>> search = session.search('mas', search_type=spotify.SearchType.SUGGEST).load()
Search(u'spotify:search:mas')
>>> search.artists[0].load().name
u'Massive Attack'
```

3.2.9 Playlist management

Another way to find some music is to use your Spotify *Playlist*, which can be found in *playlist_container*:

```
>>> len(session.playlist_container)
53
>>> playlist = session.playlist_container[0]
>>> playlist.load()
Playlist(u'spotify:user:jodal:playlist:5hBcGwxKlnzNnSrREQ4aUe')
>>> playlist.name
u'The Glitch Mob - Love Death Immortality'
```

The *PlaylistContainer* object lets you add, remove, move and rename playlists as well as playlist folders. See the API docs for *PlaylistContainer* for more examples.

```
>>> del session.playlist_container[0]
>>> len(session.playlist_container)
52
>>> session.playlist_container.insert(0, playlist)
>>> len(session.playlist_container)
53
```

The *Playlist* objects let you add, remove and move tracks in a playlist, as well as turning on things like syncing of the playlist for offline playback:

```
>>> playlist.offline_status
<PlaylistOfflineStatus.NO: 0>
>>> playlist.set_offline_mode(True)
>>> playlist.offline_status
```

(continues on next page)

(continued from previous page)

```
<PlaylistOfflineStatus.WAITING: 3>
>>> session.process_events()
# Probably needed multiple times, before syncing begins
>>> playlist.offline_status
<PlaylistOfflineStatus.DOWNLOADING: 2>
>>> playlist.offline_download_completed
20
# More process_events()
>>> playlist.offline_status
<PlaylistOfflineStatus.YES: 1>
```

For more details, see the API docs for *Playlist*.

3.2.10 Playing music

Music data is delivered to the *MUSIC_DELIVERY* event listener as PCM frames. If you want to have full control of audio playback, you can deliver these audio frames to your operating systems' audio subsystem yourself. If you want some help on the road, pyspotify comes with audio sinks for some select audio subsystems.

For details, have a look at the *spotify.AlsaSink* and *spotify.PortAudioSink* documentation, and the `examples/play_track.py` and `examples/shell.py` examples.

3.2.11 Thread safety

If you've read the libspotify documentation, you may have noticed that libspotify itself isn't thread safe. This means that you must take care to never call libspotify functions from two threads at the same time, and to finish your work with any pointers, e.g. strings, returned by libspotify functions before calling the next libspotify function. In summary, you'll need to use a single thread for all your use of libspotify, or protect all libspotify function calls with a single lock.

pyspotify, on the other hand, improves on this so that you can use pyspotify from multiple threads. pyspotify has a single global lock. This lock is acquired during all calls to libspotify, for as long as we're working with pointers returned from libspotify functions, and during all access to pyspotify's own internal state, like for example the collections of event listeners. In other words, pyspotify should be safe to use from multiple threads simultaneously.

Even though pyspotify itself is thread safe, you cannot disregard threading issues entirely when using pyspotify. There's two things to watch out for. First, event listeners for a number of the events listed in *SessionEvent* will be called from internal threads in libspotify itself. This is clearly marked in the documentation for the relevant events. Second, if you use the *EventLoop* helper thread, listeners for all other events—that is, events *not* emitted from internal threads in libspotify—will be called from the *EventLoop* thread. This shouldn't be an issue if you just use pyspotify itself from within the event listeners, but the moment you start working with your application's state from inside event listeners, you'll need to apply the proper thread synchronization primitives to avoid getting into trouble.

CHAPTER 4

API reference

4.1 API reference

The pyspotify API follows the libspotify API closely. Thus, you can refer to the similarly named functions in the libspotify docs for further details.

spotify.**__version__**
pyspotify's version number in the **PEP 386** format.

```
>>> import spotify
>>> spotify.__version__
u'2.0.0'
```

spotify.**get_libspotify_api_version**()
Get the API compatibility level of the wrapped libspotify library.

```
>>> import spotify
>>> spotify.get_libspotify_api_version()
12
```

spotify.**get_libspotify_build_id**()
Get the build ID of the wrapped libspotify library.

```
>>> import spotify
>>> spotify.get_libspotify_build_id()
u'12.1.51.g86c92b43 Release Linux-x86_64 '
```

Sections

4.1.1 Error handling

exception spotify.**Error**
A Spotify error.

This is the superclass of all custom exceptions raised by pyspotify.

classmethod maybe_raise(*error_type*, *ignores=None*)
Raise an *LibError* unless the `error_type` is `ErrorType.OK` or in the `ignores` list of error types.

Internal method.

class spotify.**ErrorType**

exception spotify.**LibError**(*error_type*)
A libspotify error.

Where many libspotify functions return error codes that must be checked after each and every function call, pyspotify raises the *LibError* exception instead. This helps you to not accidentally swallow and hide errors when using pyspotify.

error_type = None
The *ErrorType* of the error.

exception spotify.**Timeout**(*timeout*)
Exception raised by an operation not completing within the given timeout.

4.1.2 Configuration

class spotify.**Config**
The session config.

Create an instance and assign to its attributes to configure. Then use the config object to create a session:

```
>>> config = spotify.Config()
>>> config.user_agent = 'My awesome Spotify client'
>>> # Etc ...
>>> session = spotify.Session(config=config)
```

api_version
The API version of the libspotify we're using.

You should not need to change this. It defaults to the value provided by libspotify through *spotify.get_libspotify_api_version()*.

cache_location
A location for libspotify to cache files.

Defaults to `tmp` in the current working directory.

Must be a bytestring. Cannot be shared with other Spotify apps. Can only be used by one session at the time. Optimally, you should use a lock file or similar to ensure this.

settings_location
A location for libspotify to save settings.

Defaults to `tmp` in the current working directory.

Must be a bytestring. Cannot be shared with other Spotify apps. Can only be used by one session at the time. Optimally, you should use a lock file or similar to ensure this.

application_key
Your libspotify application key.

Must be a bytestring. Alternatively, you can call *load_application_key_file()*, and pyspotify will correctly read the file into *application_key*.

load_application_key_file (*filename=b'spotify_appkey.key'*)
Load your libspotify application key file.

If called without arguments, it tries to read `spotify_appkey.key` from the current working directory.

This is an alternative to setting *application_key* yourself. The file must be a binary key file, not the C code key file that can be compiled into an application.

user_agent
A string with the name of your client.

Defaults to `pyspotify 2.x.y`.

compress_playlists
Compress local copy of playlists, reduces disk space usage.

Defaults to `False`.

dont_save_metadata_for_playlists
Don't save metadata for local copies of playlists.

Defaults to `False`.

Reduces disk space usage at the expense of needing to request metadata from Spotify backend when loading list.

initially_unload_playlists
Avoid loading playlists into RAM on startup.

Defaults to `False`.

See `Playlist.in_ram()` for more details.

device_id
Device ID for offline synchronization and logging purposes.

Defaults to `None`.

The Device ID must be unique to the particular device instance, i.e. no two units must supply the same Device ID. The Device ID must not change between sessions or power cycles. Good examples is the device's MAC address or unique serial number.

Setting the device ID to an empty string has the same effect as setting it to `None`.

proxy
URL to the proxy server that should be used.

Defaults to `None`.

The format is protocol://host:port where protocol is http/https/socks4/socks5.

proxy_username
Username to authenticate with proxy server.

Defaults to `None`.

proxy_password
Password to authenticate with proxy server.

Defaults to `None`.

ca_certs_filename
Path to a file containing the root CA certificates that HTTPS servers should be verified with.

Defaults to `None`. Must be a bytestring file path otherwise.

This is not used for verifying Spotify's servers, but may be used for verifying third parties' HTTPS servers, like the Last.fm servers if you scrobbling the music you listen to through libspotify.

libspotify for OS X use other means for communicating with HTTPS servers and ignores this configuration.

The file must be a concatenation of all certificates in PEM format. Provided with libspotify is a sample PEM file in the `examples/` dir. It is recommended that the application export a similar file from the local certificate store. On Linux systems, the certificate store is often found at `/etc/ssl/certs/ca-certificates.crt` or `/etc/ssl/certs/ca-bundle.crt`

tracefile
Path to API trace file.

Defaults to `None`. Must be a bytestring otherwise.

4.1.3 Sessions

class spotify.**Session**(*config=None*)
The Spotify session.

If no `config` is provided, the default config is used.

The session object will emit a number of events. See *SessionEvent* for a list of all available events and how to connect your own listener functions up to get called when the events happens.

Warning: You can only have one *Session* instance per process. This is a libspotify limitation. If you create a second *Session* instance in the same process pyspotify will raise a `RuntimeError` with the message "Session has already been initialized".

Parameters **config** (*Config* or `None`) – the session config

config = None
A *Config* instance with the current configuration.

Once the session has been created, changing the attributes of this object will generally have no effect.

connection = None
An *Connection* instance for controlling the connection to the Spotify servers.

offline = None
An *Offline* instance for controlling offline sync.

player = None
A *Player* instance for controlling playback.

social = None
A *Social* instance for controlling social sharing.

login(*username*, *password=None*, *remember_me=False*, *blob=None*)
Authenticate to Spotify's servers.

You can login with one of two combinations:

- `username` and `password`
- `username` and `blob`

To get the `blob` string, you must once log in with `username` and `password`. You'll then get the `blob` string passed to the `credentials_blob_updated` callback.

If you set `remember_me` to `True`, you can later login to the same account without providing any `username` or credentials by calling *`relogin()`*.

logout()
Log out the current user.

If you logged in with the `remember_me` argument set to `True`, you will also need to call *`forget_me()`* to completely remove all credentials of the user that was logged in.

remembered_user_name
The username of the remembered user from a previous *`login()`* call.

relogin()
Relogin as the remembered user.

To be able do this, you must previously have logged in with *`login()`* with the `remember_me` argument set to `True`.

To check what user you'll be logged in as if you call this method, see *`remembered_user_name`*.

forget_me()
Forget the remembered user from a previous *`login()`* call.

user
The logged in *`User`*.

user_name
The username of the logged in user.

user_country
The country of the currently logged in user.

The *`OFFLINE_STATUS_UPDATED`* event is emitted on the session object when this changes.

playlist_container
The *`PlaylistContainer`* for the currently logged in user.

inbox
The inbox *`Playlist`* for the currently logged in user.

set_cache_size(*size*)
Set maximum size in MB for libspotify's cache.

If set to 0 (the default), up to 10% of the free disk space will be used.

flush_caches()
Write all cached data to disk.

libspotify does this regularly and on logout, so you should never need to call this method yourself.

preferred_bitrate(*bitrate*)
Set preferred *`Bitrate`* for music streaming.

preferred_offline_bitrate(*bitrate*, *allow_resync=False*)
Set preferred *`Bitrate`* for offline sync.

If `allow_resync` is `True` libspotify may resynchronize already synced tracks.

volume_normalization
Whether volume normalization is active or not.

Set to `True` or `False` to change.

process_events()
Process pending events in libspotify.

This method must be called for most callbacks to be called. Without calling this method, you'll only get the callbacks that are called from internal libspotify threads. When the *NOTIFY_MAIN_THREAD* event is emitted (from an internal libspotify thread), it's your job to make sure this method is called (from the thread you use for accessing Spotify), so that further callbacks can be triggered (from the same thread).

pyspotify provides an *EventLoop* that you can use for processing events when needed.

inbox_post_tracks(*canonical_username*, *tracks*, *message*, *callback=None*)

Post a `message` and one or more `tracks` to the inbox of the user with the given `canonical_username`.

`tracks` can be a single *Track* or a list of *Track* objects.

Returns an *InboxPostResult* that can be used to check if the request completed successfully.

If callback isn't `None`, it is called with an *InboxPostResult* instance when the request has completed.

get_starred(*canonical_username=None*)

Get the starred *Playlist* for the user with `canonical_username`.

If `canonical_username` isn't specified, the starred playlist for the currently logged in user is returned.

get_published_playlists(*canonical_username=None*)

Get the *PlaylistContainer* of published playlists for the user with `canonical_username`.

If `canonical_username` isn't specified, the published container for the currently logged in user is returned.

get_link(*uri*)

Get *Link* from any Spotify URI.

A link can be created from a string containing a Spotify URI on the form `spotify:...`.

Example:

```
>>> session = spotify.Session()
# ...
>>> session.get_link(
...     'spotify:track:2Foc5Q5nqNiosCNqttzHof')
Link('spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> session.get_link(
...     'http://open.spotify.com/track/4wl1dK5dHGp3Ig51stvxb0')
Link('spotify:track:4wl1dK5dHGp3Ig51stvxb0')
```

get_track(*uri*)

Get *Track* from a Spotify track URI.

Example:

```
>>> session = spotify.Session()
# ...
>>> track = session.get_track(
...     'spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> track.load().name
u'Get Lucky'
```

get_local_track(*artist=None*, *title=None*, *album=None*, *length=None*)

Get *Track* for a local track.

Spotify's official clients supports adding your local music files to Spotify so they can be played in the Spotify client. These are not synced with Spotify's servers or between your devices and there is not trace of them in your Spotify user account. The exception is when you add one of these local tracks to a playlist or mark them as starred. This creates a "local track" which pyspotify also will be able to observe.

"Local tracks" can be recognized in several ways:

- The track's URI will be of the form `spotify:local:ARTIST:ALBUM:TITLE:LENGTH_IN_SECONDS`. Any of the parts in all caps can be left out if there is no information available. That is, `spotify:local::::` is a valid local track URI.
- *`Link.type`* will be `LinkType.LOCALTRACK` for the track's link.
- *`Track.is_local`* will be `True` for the track.

This method can be used to create local tracks that can be starred or added to playlists.

`artist` may be an artist name. `title` may be a track name. `album` may be an album name. `length` may be a track length in milliseconds.

Note that when creating a local track you provide the length in milliseconds, while the local track URI contains the length in seconds.

get_album(*uri*)
Get *`Album`* from a Spotify album URI.

Example:

```
>>> session = spotify.Session()
# ...
>>> album = session.get_album(
...     'spotify:album:6wXDbHLesy6zWqQawAa91d')
>>> album.load().name
u'Forward / Return'
```

get_artist(*uri*)
Get *`Artist`* from a Spotify artist URI.

Example:

```
>>> session = spotify.Session()
# ...
>>> artist = session.get_artist(
...     'spotify:artist:22xRIphSN7IkPVbErICu7s')
>>> artist.load().name
u'Rob Dougan'
```

get_playlist(*uri*)
Get *`Playlist`* from a Spotify playlist URI.

Example:

```
>>> session = spotify.Session()
# ...
>>> playlist = session.get_playlist(
...     'spotify:user:fiat500c:playlist:54k50VZdvtnIPt4d8RBCmZ')
>>> playlist.load().name
u'500C feelgood playlist'
```

get_user(*uri*)
Get *`User`* from a Spotify user URI.

Example:

```
>>> session = spotify.Session()
# ...
```

(continues on next page)

(continued from previous page)

```
>>> user = session.get_user('spotify:user:jodal')
>>> user.load().display_name
u'jodal'
```

get_image(*uri*, *callback=None*)
Get *Image* from a Spotify image URI.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading.

Example:

```
>>> session = spotify.Session()
# ...
>>> image = session.get_image(
...     'spotify:image:a0bdcbe11b5cd126968e519b5ed1050b0e8183d0')
>>> image.load().data_uri[:50]
u''
```

search(*query*, *callback=None*, *track_offset=0*, *track_count=20*, *album_offset=0*, *album_count=20*, *artist_offset=0*, *artist_count=20*, *playlist_offset=0*, *playlist_count=20*, *search_type=None*)
Search Spotify for tracks, albums, artists, and playlists matching `query`.

> **Warning:** The search API was broken at 2016-02-03 by a server-side change made by Spotify. The functionality was never restored.
>
> Please use the Spotify Web API to perform searches.

The `query` string can be free format, or use some prefixes like `title:` and `artist:` to limit what to match on. There is no official docs on the search query format, but there's a Spotify blog post from 2008 with some examples.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, a *Search* instance, when the search completes.

The `*_offset` and `*_count` arguments can be used to retrieve more search results. libspotify will currently not respect `*_count` values higher than 200, though this may change at any time as the limit isn't documented in any official docs. If you want to retrieve more than 200 results, you'll have to search multiple times with different `*_offset` values. See the `*_total` attributes on the *Search* to see how many results exists, and to figure out how many searches you'll need to make to retrieve everything.

`search_type` is a *SearchType* value. It defaults to `SearchType.STANDARD`.

Returns a *Search* instance.

get_toplist(*type=None*, *region=None*, *canonical_username=None*, *callback=None*)
Get a *Toplist* of artists, albums, or tracks that are the currently most popular worldwide or in a specific region.

`type` is a *ToplistType* instance that specifies the type of toplist to create.

`region` is either a *ToplistRegion* instance, or a 2-letter ISO 3166-1 country code as a unicode string, that specifies the geographical region to create a toplist for.

If `region` is `ToplistRegion.USER` and `canonical_username` isn't specified, the region of the current user will be used. If `canonical_username` is specified, the region of the specified user will be used instead.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, a *Toplist* instance, when the toplist request completes.

Example:

```
>>> import spotify
>>> session = spotify.Session()
# ...
>>> toplist = session.get_toplist(
...     type=spotify.ToplistType.TRACKS, region='US')
>>> toplist.load()
>>> len(toplist.tracks)
100
>>> len(toplist.artists)
0
>>> toplist.tracks[0]
Track(u'spotify:track:2dLLR6qlu5UJ5gk0dKz0h3')
```

call(*event*, **event_args*)
Call the single registered listener for `event`.

The listener will be called with any extra arguments passed to *call()* first, and then the extra arguments passed to *on()*

Raises `AssertionError` if there is none or multiple listeners for `event`. Returns the listener's return value on success.

emit(*event*, **event_args*)
Call the registered listeners for `event`.

The listeners will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()*

num_listeners(*event=None*)
Return the number of listeners for `event`.

Return the total number of listeners for all events on this object if `event` is `None`.

off(*event=None*, *listener=None*)
Remove a `listener` that was to be called on `event`.

If `listener` is `None`, all listeners for the given `event` will be removed.

If `event` is `None`, all listeners for all events on this object will be removed.

on(*event*, *listener*, **user_args*)
Register a `listener` to be called on `event`.

The listener will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()* last.

If the listener function returns `False`, it is removed and will not be called the next time the `event` is emitted.

class spotify.**SessionEvent**
Session events.

Using the *Session* object, you can register listener functions to be called when various session related events occurs. This class enumerates the available events and the arguments your listener functions will be called with.

Example usage:

```
import spotify

def logged_in(session, error_type):
    if error_type is spotify.ErrorType.OK:
        print('Logged in as %s' % session.user)
    else:
        print('Login failed: %s' % error_type)

session = spotify.Session()
session.on(spotify.SessionEvent.LOGGED_IN, logged_in)
session.login('alice', 's3cret')
```

All events will cause debug log statements to be emitted, even if no listeners are registered. Thus, there is no need to register listener functions just to log that they're called.

LOGGED_IN = 'logged_in'
Called when login has completed.

Note that even if login has succeeded, that does not mean that you're online yet as libspotify may have cached enough information to let you authenticate with Spotify while offline.

This event should be used to get notified about login errors. To get notified about the authentication and connection state, refer to the *SessionEvent.CONNECTION_STATE_UPDATED* event.

Parameters

- **session** (*Session*) – the current session
- **error_type** (*ErrorType*) – the login error type

LOGGED_OUT = 'logged_out'
Called when logout has completed or there is a permanent connection error.

Parameters **session** (*Session*) – the current session

METADATA_UPDATED = 'metadata_updated'
Called when some metadata has been updated.

There is no way to know what metadata was updated, so you'll have to refresh all you metadata caches.

Parameters **session** (*Session*) – the current session

CONNECTION_ERROR = 'connection_error'
Called when there is a connection error and libspotify has problems reconnecting to the Spotify service.

May be called repeatedly as long as the problem persists. Will be called with an `ErrorType.OK` error when the problem is resolved.

Parameters

- **session** (*Session*) – the current session
- **error_type** (*ErrorType*) – the connection error type

MESSAGE_TO_USER = 'message_to_user'
Called when libspotify wants to show a message to the end user.

Parameters

- **session** (*Session*) – the current session
- **data** (*text*) – the message

NOTIFY_MAIN_THREAD = 'notify_main_thread'
Called when processing on the main thread is needed.

When this is called, you should call *process_events()* from your main thread. Failure to do so may cause request timeouts, or a lost connection.

> **Warning:** This event is emitted from an internal libspotify thread. Thus, your event listener must not block, and must use proper synchronization around anything it does.

Parameters **session** (*Session*) – the current session

MUSIC_DELIVERY = 'music_delivery'
Called when there is decompressed audio data available.

If the function returns a lower number of frames consumed than `num_frames`, libspotify will retry delivery of the unconsumed frames in about 100ms. This can be used for rate limiting if libspotify is giving you audio data too fast.

Note: You can register at most one event listener for this event.

> **Warning:** This event is emitted from an internal libspotify thread. Thus, your event listener must not block, and must use proper synchronization around anything it does.

Parameters

- **session** (*Session*) – the current session
- **audio_format** (*AudioFormat*) – the audio format
- **frames** (*bytestring*) – the audio frames
- **num_frames** (*int*) – the number of frames

Returns the number of frames consumed

PLAY_TOKEN_LOST = 'play_token_lost'
Music has been paused because an account only allows music to be played from one location simultaneously.

When this event is emitted, you should pause playback.

Parameters **session** (*Session*) – the current session

LOG_MESSAGE = 'log_message'
Called when libspotify have something to log.

Note that pyspotify logs this for you, so you'll probably never need to register a listener for this event.

Parameters

- **session** (*Session*) – the current session
- **data** (*text*) – the message

END_OF_TRACK = 'end_of_track'
Called when all audio data for the current track has been delivered.

Parameters **session** (*Session*) – the current session

STREAMING_ERROR = 'streaming_error'
Called when audio streaming cannot start or continue.

Parameters

- **session** (*Session*) – the current session
- **error_type** (*ErrorType*) – the streaming error type

USER_INFO_UPDATED = 'user_info_updated'
Called when anything related to *User* objects is updated.

Parameters **session** (*Session*) – the current session

START_PLAYBACK = 'start_playback'
Called when audio playback should start.

You need to implement a listener for the *GET_AUDIO_BUFFER_STATS* event for the *START_PLAYBACK* event to be useful.

> **Warning:** This event is emitted from an internal libspotify thread. Thus, your event listener must not block, and must use proper synchronization around anything it does.

Parameters **session** (*Session*) – the current session

STOP_PLAYBACK = 'stop_playback'
Called when audio playback should stop.

You need to implement a listener for the *GET_AUDIO_BUFFER_STATS* event for the *STOP_PLAYBACK* event to be useful.

> **Warning:** This event is emitted from an internal libspotify thread. Thus, your event listener must not block, and must use proper synchronization around anything it does.

Parameters **session** (*Session*) – the current session

GET_AUDIO_BUFFER_STATS = 'get_audio_buffer_stats'
Called to query the application about its audio buffer.

Note: You can register at most one event listener for this event.

> **Warning:** This event is emitted from an internal libspotify thread. Thus, your event listener must not block, and must use proper synchronization around anything it does.

Parameters **session** (*Session*) – the current session

Returns an *AudioBufferStats* instance

OFFLINE_STATUS_UPDATED = 'offline_status_updated'
Called when offline sync status is updated.

Parameters **session** (*Session*) – the current session

CREDENTIALS_BLOB_UPDATED = 'credentials_blob_updated'
Called when storable credentials have been updated, typically right after login.

The `blob` argument can be stored and later passed to *login()* to login without storing the user's password.

Parameters

- **session** (*Session*) – the current session
- **blob** (*bytestring*) – the authentication blob

CONNECTION_STATE_UPDATED = 'connection_state_updated'
Called when the connection state is updated.

The connection state includes login, logout, offline mode, etc.

Parameters **session** (*Session*) – the current session

SCROBBLE_ERROR = 'scrobble_error'
Called when there is a scrobble error event.

Parameters

- **session** (*Session*) – the current session
- **error_type** (*ErrorType*) – the scrobble error type

PRIVATE_SESSION_MODE_CHANGED = 'private_session_mode_changed'
Called when there is a change in the private session mode.

Parameters

- **session** (*Session*) – the current session
- **is_private** (*bool*) – whether the session is private

4.1.4 Event loop

class spotify.**EventLoop**(*session*)
Event loop for automatically processing events from libspotify.

The event loop is a `Thread` that listens to *NOTIFY_MAIN_THREAD* events and calls *process_events()* when needed.

To use it, pass it your *Session* instance and call *start()*:

```
>>> session = spotify.Session()
>>> event_loop = spotify.EventLoop(session)
>>> event_loop.start()
```

The event loop thread is a daemon thread, so it will not stop your application from exiting. If you wish to stop the event loop without stopping your entire application, call *stop()*. You may call `join()` to block until the event loop thread has finished, just like for any other thread.

Warning: If you use *EventLoop* to process the libspotify events, any event listeners you've registered will be called from the event loop thread. pyspotify itself is thread safe, but you'll need to ensure that you have proper synchronization in your own application code, as always when working with threads.

start()
Start the event loop.

stop()
Stop the event loop.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

4.1.5 Connection

class spotify.connection.**Connection**(*session*)
Connection controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the `connection` attribute on the `Session` instance.

state
The session's current `ConnectionState`.

The connection state involves two components, authentication and offline mode. The mapping is as follows

- `LOGGED_OUT`: not authenticated, offline
- `OFFLINE`: authenticated, offline
- `LOGGED_IN`: authenticated, online
- `DISCONNECTED`: authenticated, offline, was previously online

Register listeners for the *spotify.SessionEvent.CONNECTION_STATE_UPDATED* event to be notified when the connection state changes.

type
The session's `ConnectionType`.

Defaults to `ConnectionType.UNKNOWN`. Set to a `ConnectionType` value to tell libspotify what type of connection you're using.

This is used together with *allow_network*, *allow_network_if_roaming*, *allow_sync_over_wifi*, and *allow_sync_over_mobile* to control offline syncing and network usage.

allow_network
Whether or not network access is allowed at all.

Defaults to `True`. Setting this to `False` turns on offline mode.

allow_network_if_roaming
Whether or not network access is allowed if *type* is set to `ConnectionType.MOBILE_ROAMING`.

Defaults to `False`.

allow_sync_over_wifi
Whether or not offline syncing is allowed when *type* is set to `ConnectionType.WIFI`.

Defaults to `True`.

allow_sync_over_mobile
Whether or not offline syncing is allowed when *type* is set to `ConnectionType.MOBILE`, or *allow_network_if_roaming* is `True` and *type* is set to `ConnectionType.MOBILE_ROAMING`.

Defaults to `True`.

class spotify.**ConnectionRule**

class spotify.**ConnectionState**

class spotify.**ConnectionType**

4.1.6 Offline sync

class spotify.offline.**Offline**(*session*)
Offline sync controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the `offline` attribute on the `Session` instance.

tracks_to_sync
Total number of tracks that needs download before everything from all playlists that are marked for offline is fully synchronized.

num_playlists
Number of playlists that are marked for offline synchronization.

sync_status
The `OfflineSyncStatus` or `None` if not syncing.

The `OFFLINE_STATUS_UPDATED` event is emitted on the session object when this is updated.

time_left
The number of seconds until the user has to get online and relogin.

class spotify.**OfflineSyncStatus**(*sp_offline_sync_status*)
A Spotify offline sync status object.

You'll never need to create an instance of this class yourself. You'll find it ready for use as the `sync_status` attribute on the *`offline`* attribute on the *`Session`* instance.

queued_tracks
Number of tracks left to sync in current sync operation.

done_tracks
Number of tracks marked for sync that existed on the device before the current sync operation.

copied_tracks
Number of tracks copied during the current sync operation.

willnotcopy_tracks
Number of tracks marked for sync that will not be copied.

error_tracks
Number of tracks that failed syncing during the current sync operation.

syncing
If sync operation is in progress.

4.1.7 Links

class spotify.**Link**(*session*, *uri=None*, *sp_link=None*, *add_ref=True*)
A Spotify object link.

Call the *get_link()* method on your *Session* instance to get a *Link* object from a Spotify URI. You can also get links from the `link` attribute on most objects, e.g. *Track.link*.

To get the URI from the link object you can use the *uri* attribute, or simply use the link as a string:

```
>>> session = spotify.Session()
# ...
>>> link = session.get_link(
...     'spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> link
Link('spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> link.uri
'spotify:track:2Foc5Q5nqNiosCNqttzHof'
>>> str(link)
'spotify:track:2Foc5Q5nqNiosCNqttzHof'
>>> link.type
<LinkType.TRACK: 1>
>>> track = link.as_track()
>>> track.link
Link('spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> track.load().name
u'Get Lucky'
```

You can also get *Link* objects from open.spotify.com and play.spotify.com URLs:

```
>>> session.get_link(
...     'http://open.spotify.com/track/4wl1dK5dHGp3Ig51stvxb0')
Link('spotify:track:4wl1dK5dHGp3Ig51stvxb0')
>>> session.get_link(
...     'https://play.spotify.com/track/4wl1dK5dHGp3Ig51stvxb0'
...     '?play=true&utm_source=open.spotify.com&utm_medium=open')
Link('spotify:track:4wl1dK5dHGp3Ig51stvxb0')
```

uri
: The link's Spotify URI.

url
: The link's HTTP URL.

type
: The link's *LinkType*.

as_track()
: Make a *Track* from the link.

as_track_offset()
: Get the track offset in milliseconds from the link.

as_album()
: Make an *Album* from the link.

as_artist()
: Make an *Artist* from the link.

as_playlist()
: Make a *Playlist* from the link.

as_user()
: Make an *User* from the link.

as_image(*callback=None*)
Make an *Image* from the link.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading.

class spotify.**LinkType**

4.1.8 Users

class spotify.**User**(*session*, *uri=None*, *sp_user=None*, *add_ref=True*)
A Spotify user.

You can get users from the session, or you can create a *User* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> user = session.get_user('spotify:user:jodal')
>>> user.load().display_name
u'jodal'
```

canonical_name
The user's canonical username.

display_name
The user's displayable username.

is_loaded
Whether the user's data is loaded yet.

load(*timeout=None*)
Block until the user's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

link
A *Link* to the user.

starred
The *Playlist* of tracks starred by the user.

published_playlists
The *PlaylistContainer* of playlists published by the user.

4.1.9 Tracks

class spotify.**Track**(*session*, *uri=None*, *sp_track=None*, *add_ref=True*)
A Spotify track.

You can get tracks from playlists or albums, or you can create a *Track* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> track = session.get_track('spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> track.load().name
u'Get Lucky'
```

is_loaded
Whether the track's data is loaded.

error
An *ErrorType* associated with the track.

Check to see if there was problems loading the track.

load(*timeout=None*)
Block until the track's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

offline_status
The *TrackOfflineStatus* of the track.

The `metadata_updated` callback is called when the offline status changes.

Will always return `None` if the track isn't loaded.

availability
The *TrackAvailability* of the track.

Will always return `None` if the track isn't loaded.

is_local
Whether the track is a local track.

Will always return `None` if the track isn't loaded.

is_autolinked
Whether the track is a autolinked to another track.

Will always return `None` if the track isn't loaded.

See *playable*.

playable
The actual track that will be played when this track is played.

Will always return `None` if the track isn't loaded.

See *is_autolinked*.

is_placeholder
Whether the track is a placeholder for a non-track object in the playlist.

To convert to the real object:

```
>>> session = spotify.Session()
# ...
>>> track = session.get_track(
...     'spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> track.load()
>>> track.is_placeholder
True
>>> track.link.type
<LinkType.ARTIST: ...>
>>> artist = track.link.as_artist()
```

Will always return `None` if the track isn't loaded.

starred
Whether the track is starred by the current user.

Set to `True` or `False` to change.

Will always be `None` if the track isn't loaded.

artists
The artists performing on the track.

Will always return an empty list if the track isn't loaded.

album
The album of the track.

Will always return `None` if the track isn't loaded.

name
The track's name.

Will always return `None` if the track isn't loaded.

duration
The track's duration in milliseconds.

Will always return `None` if the track isn't loaded.

popularity
The track's popularity in the range 0-100, 0 if undefined.

Will always return `None` if the track isn't loaded.

disc
The track's disc number. 1 or higher.

Will always return `None` if the track isn't part of an album or artist browser.

index
The track's index number. 1 or higher.

Will always return `None` if the track isn't part of an album or artist browser.

link
A *Link* to the track.

link_with_offset(*offset*)
A *Link* to the track with an `offset` in milliseconds into the track.

class spotify.**TrackAvailability**

class spotify.**TrackOfflineStatus**

4.1.10 Albums

class spotify.**Album**(*session*, *uri=None*, *sp_album=None*, *add_ref=True*)
A Spotify album.

You can get an album from a track or an artist, or you can create an *Album* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> album = session.get_album('spotify:album:6wXDbHLesy6zWqQawAa91d')
>>> album.load().name
u'Forward / Return'
```

is_loaded
Whether the album's data is loaded.

load(*timeout=None*)
Block until the album's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

is_available
Whether the album is available in the current region.

Will always return `None` if the album isn't loaded.

artist
The artist of the album.

Will always return `None` if the album isn't loaded.

cover(*image_size=None*, *callback=None*)
The album's cover *Image*.

`image_size` is an *ImageSize* value, by default `ImageSize.NORMAL`.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading.

Will always return `None` if the album isn't loaded or the album has no cover.

cover_link(*image_size=None*)
A *Link* to the album's cover.

`image_size` is an *ImageSize* value, by default `ImageSize.NORMAL`.

This is equivalent with `album.cover(image_size).link`, except that this method does not need to create the album cover image object to create a link to it.

name
The album's name.

Will always return `None` if the album isn't loaded.

year
The album's release year.

Will always return `None` if the album isn't loaded.

type
The album's *AlbumType*.

Will always return `None` if the album isn't loaded.

link
A *Link* to the album.

browse(*callback=None*)
Get an *AlbumBrowser* for the album.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *AlbumBrowser* instance, when the browser is done loading.

Can be created without the album being loaded.

class spotify.**AlbumBrowser**(*session*, *album=None*, *callback=None*, *sp_albumbrowse=None*, *add_ref=True*)

An album browser for a Spotify album.

You can get an album browser from any *Album* instance by calling *Album.browse()*:

```
>>> session = spotify.Session()
# ...
>>> album = session.get_album('spotify:album:6wXDbHLesy6zWqQawAa91d')
>>> browser = album.browse()
>>> browser.load()
>>> len(browser.tracks)
7
```

loaded_event = None

threading.Event that is set when the album browser is loaded.

is_loaded

Whether the album browser's data is loaded.

load(*timeout=None*)

Block until the album browser's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

error

An *ErrorType* associated with the album browser.

Check to see if there was problems creating the album browser.

backend_request_duration

The time in ms that was spent waiting for the Spotify backend to create the album browser.

Returns `-1` if the request was served from local cache. Returns `None` if the album browser isn't loaded yet.

album

Get the *Album* the browser is for.

Will always return `None` if the album isn't loaded.

artist

The *Artist* of the album.

Will always return `None` if the album isn't loaded.

copyrights

The album's copyright strings.

Will always return an empty list if the album browser isn't loaded.

tracks

The album's tracks.

Will always return an empty list if the album browser isn't loaded.

review

A review of the album.

Will always return an empty string if the album browser isn't loaded.

class spotify.**AlbumType**

4.1.11 Artists

class spotify.**Artist**(*session*, *uri=None*, *sp_artist=None*, *add_ref=True*)
A Spotify artist.

You can get artists from tracks and albums, or you can create an *Artist* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> artist = session.get_artist(
...     'spotify:artist:22xRIphSN7IkPVbErICu7s')
>>> artist.load().name
u'Rob Dougan'
```

name
The artist's name.

Will always return `None` if the artist isn't loaded.

is_loaded
Whether the artist's data is loaded.

load(*timeout=None*)
Block until the artist's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

portrait(*image_size=None*, *callback=None*)
The artist's portrait *Image*.

`image_size` is an *ImageSize* value, by default `ImageSize.NORMAL`.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading.

Will always return `None` if the artist isn't loaded or the artist has no portrait.

portrait_link(*image_size=None*)
A *Link* to the artist's portrait.

`image_size` is an *ImageSize* value, by default `ImageSize.NORMAL`.

This is equivalent with `artist.portrait(image_size).link`, except that this method does not need to create the artist portrait image object to create a link to it.

link
A *Link* to the artist.

browse(*type=None*, *callback=None*)
Get an *ArtistBrowser* for the artist.

If `type` is `None`, it defaults to `ArtistBrowserType.FULL`.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *ArtistBrowser* instance, when the browser is done loading.

Can be created without the artist being loaded.

class spotify.**ArtistBrowser**(*session*, *artist=None*, *type=None*, *callback=None*, *sp_artistbrowse=None*, *add_ref=True*)
An artist browser for a Spotify artist.

You can get an artist browser from any *Artist* instance by calling *Artist.browse()*:

```
>>> session = spotify.Session()
# ...
>>> artist = session.get_artist(
...     'spotify:artist:421vyBBkhgRAOz4cYPvrZJ')
>>> browser = artist.browse()
>>> browser.load()
>>> len(browser.albums)
7
```

loaded_event = None
threading.Event that is set when the artist browser is loaded.

is_loaded
Whether the artist browser's data is loaded.

load(*timeout=None*)
Block until the artist browser's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

error
An *ErrorType* associated with the artist browser.

Check to see if there was problems creating the artist browser.

backend_request_duration
The time in ms that was spent waiting for the Spotify backend to create the artist browser.

Returns `-1` if the request was served from local cache. Returns `None` if the artist browser isn't loaded yet.

artist
Get the *Artist* the browser is for.

Will always return `None` if the artist browser isn't loaded.

portraits(*callback=None*)
The artist's portraits.

Due to limitations in libspotify's API you can't specify the *ImageSize* of these images.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading. The callable will be called once for each portrait.

Will always return an empty list if the artist browser isn't loaded.

tracks
The artist's tracks.

Will be an empty list if the browser was created with a `type` of `ArtistBrowserType.NO_TRACKS` or `ArtistBrowserType.NO_ALBUMS`.

Will always return an empty list if the artist browser isn't loaded.

tophit_tracks
The artist's top hit tracks.

Will always return an empty list if the artist browser isn't loaded.

albums
The artist's albums.

Will be an empty list if the browser was created with a `type` of `ArtistBrowserType.NO_ALBUMS`.

Will always return an empty list if the artist browser isn't loaded.

similar_artists
The artist's similar artists.

Will always return an empty list if the artist browser isn't loaded.

biography
A biography of the artist.

Will always return an empty string if the artist browser isn't loaded.

class spotify.**ArtistBrowserType**

4.1.12 Images

class spotify.**Image**(*session*, *uri=None*, *sp_image=None*, *add_ref=True*, *callback=None*)
A Spotify image.

You can get images from *Album.cover()*, *Artist.portrait()*, *Playlist.image()*, or you can create an *Image* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> image = session.get_image(
...     'spotify:image:a0bdcbe11b5cd126968e519b5ed1050b0e8183d0')
>>> image.load().data_uri[:50]
u''
```

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading.

loaded_event = None
`threading.Event` that is set when the image is loaded.

is_loaded
Whether the image's data is loaded.

error
An *ErrorType* associated with the image.

Check to see if there was problems loading the image.

load(*timeout=None*)
Block until the image's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

format
The *ImageFormat* of the image.

Will always return `None` if the image isn't loaded.

data
The raw image data as a bytestring.

Will always return `None` if the image isn't loaded.

data_uri
The raw image data as a data: URI.

Will always return `None` if the image isn't loaded.

link
A *Link* to the image.

class spotify.**ImageFormat**

class spotify.**ImageSize**

4.1.13 Search

Warning: The search API was broken at 2016-02-03 by a server-side change made by Spotify. The functionality was never restored.

Please use the Spotify Web API to perform searches.

class spotify.**Search**(*session*, *query=''*, *callback=None*, *track_offset=0*, *track_count=20*, *album_offset=0*, *album_count=20*, *artist_offset=0*, *artist_count=20*, *playlist_offset=0*, *playlist_count=20*, *search_type=None*, *sp_search=None*, *add_ref=True*)
A Spotify search result.

Call the *search()* method on your *Session* instance to do a search and get a *Search* back.

loaded_event = None
`threading.Event` that is set when the search is loaded.

is_loaded
Whether the search's data is loaded.

error
An *ErrorType* associated with the search.

Check to see if there was problems loading the search.

load(*timeout=None*)
Block until the search's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

query
The search query.

Will always return `None` if the search isn't loaded.

did_you_mean
The search's "did you mean" query or `None` if no such suggestion exists.

Will always return `None` if the search isn't loaded.

tracks
The tracks matching the search query.

Will always return an empty list if the search isn't loaded.

track_total
The total number of tracks matching the search query.

If the number is larger than the interval specified at search object creation, more search results are available. To fetch these, create a new search object with a new interval.

albums
The albums matching the search query.

Will always return an empty list if the search isn't loaded.

album_total
The total number of albums matching the search query.

If the number is larger than the interval specified at search object creation, more search results are available. To fetch these, create a new search object with a new interval.

artists
The artists matching the search query.

Will always return an empty list if the search isn't loaded.

artist_total
The total number of artists matching the search query.

If the number is larger than the interval specified at search object creation, more search results are available. To fetch these, create a new search object with a new interval.

playlists
The playlists matching the search query as *SearchPlaylist* objects containing the name, URI and image URI for matching playlists.

Will always return an empty list if the search isn't loaded.

playlist_total
The total number of playlists matching the search query.

If the number is larger than the interval specified at search object creation, more search results are available. To fetch these, create a new search object with a new interval.

more(*callback=None*, *track_count=None*, *album_count=None*, *artist_count=None*, *playlist_count=None*)
Get the next page of search results for the same query.

If called without arguments, the `callback` and `*_count` arguments from the original search is reused. If anything other than `None` is specified, the value is used instead.

link
A *Link* to the search.

class spotify.**SearchPlaylist**(*session*, *name*, *uri*, *image_uri*)
A playlist matching a search query.

name = None
The name of the playlist.

uri = None
The URI of the playlist.

image_uri = None
The URI of the playlist's image.

playlist
The *Playlist* object for this *SearchPlaylist*.

image
The *Image* object for this *SearchPlaylist*.

class spotify.**SearchType**

4.1.14 Playlists

class spotify.**Playlist**(*session*, *uri=None*, *sp_playlist=None*, *add_ref=True*)
A Spotify playlist.

You can get playlists from the *playlist_container*, *inbox*, *get_starred()*, *search()*, etc., or you can create a playlist yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> playlist = session.get_playlist(
...     'spotify:user:fiat500c:playlist:54k50VZdvtnIPt4d8RBCmZ')
>>> playlist.load().name
u'500C feelgood playlist'
```

is_loaded
Whether the playlist's data is loaded.

load(*timeout=None*)
Block until the playlist's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

tracks
The playlist's tracks.

Will always return an empty list if the playlist isn't loaded.

tracks_with_metadata
The playlist's tracks, with metadata specific to the playlist as a a list of *PlaylistTrack* objects.

Will always return an empty list if the playlist isn't loaded.

name
The playlist's name.

Assigning to *name* will rename the playlist.

Will always return `None` if the playlist isn't loaded.

rename(*new_name*)
Rename the playlist.

owner
The *User* object for the owner of the playlist.

collaborative
Whether the playlist can be modified by all users or not.

Set to `True` or `False` to change.

set_autolink_tracks(*link=True*)
If a playlist is autolinked, unplayable tracks will be made playable by linking them to other Spotify tracks, where possible.

description
The playlist's description.

Will return `None` if the description is unset.

image(*callback=None*)
The playlist's *Image*.

Due to limitations in libspotify's API you can't specify the *ImageSize* of these images.

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading.

Will always return `None` if the playlist isn't loaded or the playlist has no image.

has_pending_changes
Check if the playlist has local changes that has not been acknowledged by the server yet.

add_tracks(*tracks*, *index=None*)
Add the given `tracks` to playlist at the given `index`.

`tracks` can either be a single *Track* or a list of *Track* objects. If `index` isn't specified, the tracks are added to the end of the playlist.

remove_tracks(*indexes*)
Remove the tracks at the given `indexes` from the playlist.

`indexes` can be a single index or a list of indexes to remove.

reorder_tracks(*indexes*, *new_index*)
Move the tracks at the given `indexes` to a `new_index` in the playlist.

`indexes` can be a single index or a list of indexes to move.

`new_index` must be equal to or lower than the current playlist length.

num_subscribers
The number of subscribers to the playlist.

The number can be higher than the length of the *subscribers* collection, especially if the playlist got many subscribers.

May be zero until you call *update_subscribers()* and the *SUBSCRIBERS_CHANGED* event is emitted from the playlist.

subscribers
The canonical usernames of up to 500 of the subscribers of the playlist.

May be empty until you call *update_subscribers()* and the *SUBSCRIBERS_CHANGED* event is emitted from the playlist.

update_subscribers()
Request an update of *num_subscribers* and the *subscribers* collection.

The *SUBSCRIBERS_CHANGED* event is emitted from the playlist when the subscriber data has been updated.

is_in_ram
Whether the playlist is in RAM, and not only on disk.

A playlist must *currently be* in RAM for tracks to be available. A playlist must *have been* in RAM for other metadata to be available.

By default, playlists are kept in RAM unless *initially_unload_playlists* is set to `True` before creating the *Session*. If the playlists are initially unloaded, use *set_in_ram()* to have a playlist loaded into RAM.

set_in_ram(*in_ram=True*)
Control whether or not to keep the playlist in RAM.

See *is_in_ram* for further details.

set_offline_mode(*offline=True*)
Mark the playlist to be synchronized for offline playback.

The playlist must be in the current user's playlist container.

offline_status
The playlist's *PlaylistOfflineStatus*.

offline_download_completed
The download progress for an offline playlist.

A number in the range 0-100. Always `None` if *offline_status* isn't `PlaylistOfflineStatus.DOWNLOADING`.

link
A *Link* to the playlist.

on(*event*, *listener*, **user_args*)
Register a `listener` to be called on `event`.

The listener will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()* last.

If the listener function returns `False`, it is removed and will not be called the next time the `event` is emitted.

off(*event=None*, *listener=None*)
Remove a `listener` that was to be called on `event`.

If `listener` is `None`, all listeners for the given `event` will be removed.

If `event` is `None`, all listeners for all events on this object will be removed.

call(*event*, **event_args*)
Call the single registered listener for `event`.

The listener will be called with any extra arguments passed to *call()* first, and then the extra arguments passed to *on()*

Raises `AssertionError` if there is none or multiple listeners for `event`. Returns the listener's return value on success.

emit(*event*, **event_args*)
Call the registered listeners for `event`.

The listeners will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()*

num_listeners(*event=None*)
Return the number of listeners for `event`.

Return the total number of listeners for all events on this object if `event` is `None`.

class spotify.**PlaylistEvent**

Playlist events.

Using *Playlist* objects, you can register listener functions to be called when various events occurs in the playlist. This class enumerates the available events and the arguments your listener functions will be called with.

Example usage:

```
import spotify

def tracks_added(playlist, tracks, index):
    print('Tracks added to playlist')

session = spotify.Session()
# Login, etc...

playlist = session.playlist_container[0]
playlist.on(spotify.PlaylistEvent.TRACKS_ADDED, tracks_added)
```

All events will cause debug log statements to be emitted, even if no listeners are registered. Thus, there is no need to register listener functions just to log that they're called.

TRACKS_ADDED = 'tracks_added'

Called when one or more tracks have been added to the playlist.

Parameters

- **playlist** (*Playlist*) – the playlist
- **tracks** (list of *Track*) – the added tracks
- **index** (*int*) – the index in the playlist the tracks were added at

TRACKS_REMOVED = 'tracks_removed'

Called when one or more tracks have been removed from the playlist.

Parameters

- **playlist** (*Playlist*) – the playlist
- **indexes** (*list of ints*) – indexes of the tracks that were removed

TRACKS_MOVED = 'tracks_moved'

Called when one or more tracks have been moved within a playlist.

Parameters

- **playlist** (*Playlist*) – the playlist
- **old_indexes** (*list of ints*) – old indexes of the tracks that were moved
- **new_index** (*int*) – the new index in the playlist the tracks were moved to

PLAYLIST_RENAMED = 'playlist_renamed'

Called when the playlist has been renamed.

Parameters **playlist** (*Playlist*) – the playlist

PLAYLIST_STATE_CHANGED = 'playlist_state_changed'

Called when the state changed for a playlist.

There are three states that trigger this callback:

- Collaboration for this playlist has been turned on or off. See `Playlist.is_collaborative()`.

- The playlist started having pending changes, or all pending changes have now been committed. See *Playlist.has_pending_changes*.
- The playlist started loading, or finished loading. See *Playlist.is_loaded*.

Parameters **playlist** (*Playlist*) – the playlist

PLAYLIST_UPDATE_IN_PROGRESS = 'playlist_update_in_progress'
Called when a playlist is updating or is done updating.

This is called before and after a series of changes are applied to the playlist. It allows e.g. the user interface to defer updating until the entire operation is complete.

Parameters

- **playlist** (*Playlist*) – the playlist
- **done** (*bool*) – if the update is completed

PLAYLIST_METADATA_UPDATED = 'playlist_metadata_updated'
Called when metadata for one or more tracks in the playlist have been updated.

Parameters **playlist** (*Playlist*) – the playlist

TRACK_CREATED_CHANGED = 'track_created_changed'
Called when the create time and/or creator for a playlist entry changes.

Parameters

- **playlist** (*Playlist*) – the playlist
- **index** (*int*) – the index of the entry in the playlist that was changed
- **user** (*User*) – the user that created the playlist entry
- **time** (*int*) – the time the entry was created, in seconds since Unix epoch

TRACK_SEEN_CHANGED = 'track_seen_changed'
Called when the seen attribute of a playlist entry changes.

Parameters

- **playlist** (*Playlist*) – the playlist
- **index** (*int*) – the index of the entry in the playlist that was changed
- **seen** (*bool*) – whether the entry is seen or not

DESCRIPTION_CHANGED = 'description_changed'
Called when the playlist description has changed.

Parameters

- **playlist** (*Playlist*) – the playlist
- **description** (*string*) – the new description

IMAGE_CHANGED = 'image_changed'
Called when the playlist image has changed.

Parameters

- **playlist** (*Playlist*) – the playlist
- **image** (*Image*) – the new image

TRACK_MESSAGE_CHANGED = 'track_message_changed'
Called when the message attribute of a playlist entry changes.

Parameters

- **playlist** (*Playlist*) – the playlist
- **index** (*int*) – the index of the entry in the playlist that was changed
- **message** (*string*) – the new message

SUBSCRIBERS_CHANGED = 'subscribers_changed'
Called when playlist subscribers changes, either the count or the subscriber names.

Parameters **playlist** (*Playlist*) – the playlist

class spotify.**PlaylistContainer**(*session*, *sp_playlistcontainer*, *add_ref=True*)
A Spotify playlist container.

The playlist container can be accessed as a regular Python collection to work with the playlists:

```
>>> import spotify
>>> session = spotify.Session()
# Login, etc.
>>> container = session.playlist_container
>>> container.is_loaded
False
>>> container.load()
[Playlist(u'spotify:user:jodal:playlist:6xkJysqhkj9uwufFbUb8sP'),
 Playlist(u'spotify:user:jodal:playlist:0agJjPcOhHnstLIQunJHxo'),
 PlaylistFolder(id=8027491506140518932L, name=u'Shared playlists',
    type=<PlaylistType.START_FOLDER: 1>),
 Playlist(u'spotify:user:p3.no:playlist:7DkMndS2KNVQuf2fOpMt10'),
 PlaylistFolder(id=8027491506140518932L, name=u'',
    type=<PlaylistType.END_FOLDER: 2>)]
>>> container[0]
Playlist(u'spotify:user:jodal:playlist:6xkJysqhkj9uwufFbUb8sP')
```

As you can see, a playlist container can contain a mix of *Playlist* and *PlaylistFolder* objects.

The container supports operations that changes the container as well.

To add a playlist you can use *append()* or *insert()* with either the name of a new playlist or an existing playlist object. For example:

```
>>> playlist = session.get_playlist(
...     'spotify:user:fiat500c:playlist:54k50VZdvtnIPt4d8RBCmZ')
>>> container.insert(3, playlist)
>>> container.append('New empty playlist')
```

To remove a playlist or folder you can use *remove_playlist()*, or:

```
>>> del container[0]
```

To replace an existing playlist or folder with a new empty playlist with the given name you can use *remove_playlist()* and *add_new_playlist()*, or:

```
>>> container[0] = 'My other new empty playlist'
```

To replace an existing playlist or folder with an existing playlist you can :use *remove_playlist()* and *add_playlist()*, or:

```
>>> container[0] = playlist
```

is_loaded
Whether the playlist container's data is loaded.

load(*timeout=None*)
Block until the playlist container's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

add_new_playlist(*name*, *index=None*)
Add an empty playlist with `name` at the given `index`.

The playlist name must not be space-only or longer than 255 chars.

If the `index` isn't specified, the new playlist is added at the end of the container.

Returns the new playlist.

add_playlist(*playlist*, *index=None*)
Add an existing `playlist` to the playlist container at the given `index`.

The playlist can either be a *Playlist*, or a *Link* linking to a playlist.

If the `index` isn't specified, the playlist is added at the end of the container.

Returns the added playlist, or `None` if the playlist already existed in the container. If the playlist already exists, it will not be moved to the given `index`.

add_folder(*name*, *index=None*)
Add a playlist folder with `name` at the given `index`.

The playlist folder name must not be space-only or longer than 255 chars.

If the `index` isn't specified, the folder is added at the end of the container.

remove_playlist(*index*, *recursive=False*)
Remove playlist at the given index from the container.

If the item at the given `index` is the start or the end of a playlist folder, and the other end of the folder is found, it is also removed. The folder content is kept, but is moved one level up the folder hierarchy. If `recursive` is `True`, the folder content is removed as well.

Using `del playlist_container[3]` is equivalent to `playlist_container.remove_playlist(3)`. Similarly, `del playlist_container[0:2]` is equivalent to calling this method with indexes `1` and `0`.

move_playlist(*from_index*, *to_index*, *dry_run=False*)
Move playlist at `from_index` to `to_index`.

If `dry_run` is `True` the move isn't actually done. It is only checked if the move is possible.

owner
The *User* object for the owner of the playlist container.

get_unseen_tracks(*playlist*)
Get a list of unseen tracks in the given `playlist`.

The list is a *PlaylistUnseenTracks* instance.

The tracks will remain "unseen" until *clear_unseen_tracks()* is called on the playlist.

clear_unseen_tracks(*playlist*)
Clears unseen tracks from the given `playlist`.

insert(*index*, *value*)
S.insert(index, value) – insert value before index

on(*event*, *listener*, **user_args*)
Register a `listener` to be called on `event`.

The listener will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()* last.

If the listener function returns `False`, it is removed and will not be called the next time the `event` is emitted.

off(*event=None*, *listener=None*)
Remove a `listener` that was to be called on `event`.

If `listener` is `None`, all listeners for the given `event` will be removed.

If `event` is `None`, all listeners for all events on this object will be removed.

append(*value*)
S.append(value) – append value to the end of the sequence

call(*event*, **event_args*)
Call the single registered listener for `event`.

The listener will be called with any extra arguments passed to *call()* first, and then the extra arguments passed to *on()*

Raises `AssertionError` if there is none or multiple listeners for `event`. Returns the listener's return value on success.

clear() → None – remove all items from S

count(*value*) → integer – return number of occurrences of value

emit(*event*, **event_args*)
Call the registered listeners for `event`.

The listeners will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()*

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

num_listeners(*event=None*)
Return the number of listeners for `event`.

Return the total number of listeners for all events on this object if `event` is `None`.

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

class spotify.**PlaylistContainerEvent**
Playlist container events.

Using *PlaylistContainer* objects, you can register listener functions to be called when various events occurs in the playlist container. This class enumerates the available events and the arguments your listener functions will be called with.

Example usage:

```
import spotify

def container_loaded(playlist_container):
    print('Playlist container loaded')

session = spotify.Session()
# Login, etc...

session.playlist_container.on(
    spotify.PlaylistContainerEvent.CONTAINER_LOADED, container_loaded)
```

All events will cause debug log statements to be emitted, even if no listeners are registered. Thus, there is no need to register listener functions just to log that they're called.

PLAYLIST_ADDED = 'playlist_added'
Called when a playlist is added to the container.

Parameters

- **playlist_container** (*PlaylistContainer*) – the playlist container
- **playlist** (*Playlist*) – the added playlist
- **index** (*int*) – the index the playlist was added at

PLAYLIST_REMOVED = 'playlist_removed'
Called when a playlist is removed from the container.

Parameters

- **playlist_container** (*PlaylistContainer*) – the playlist container
- **playlist** (*Playlist*) – the removed playlist
- **index** (*int*) – the index the playlist was removed from

PLAYLIST_MOVED = 'playlist_moved'
Called when a playlist is moved in the container.

Parameters

- **playlist_container** (*PlaylistContainer*) – the playlist container
- **playlist** (*Playlist*) – the moved playlist
- **old_index** (*int*) – the index the playlist was moved from
- **new_index** (*int*) – the index the playlist was moved to

CONTAINER_LOADED = 'container_loaded'
Called when the playlist container is loaded.

Parameters **playlist_container** (*PlaylistContainer*) – the playlist container

class spotify.**PlaylistFolder**
An object marking the start or end of a playlist folder.

id
An opaque ID that matches the ID of the *PlaylistFolder* object at the other end of the folder.

name
Name of the playlist folder. This is an empty string for the END_FOLDER.

type
The *PlaylistType* of the folder. Either START_FOLDER or END_FOLDER.

class spotify.**PlaylistOfflineStatus**

class spotify.**PlaylistPlaceholder**
An object marking an unknown entry in the playlist container.

class spotify.**PlaylistTrack**(*session*, *sp_playlist*, *index*)
A playlist track with metadata specific to the playlist.

Use *tracks_with_metadata* to get a list of *PlaylistTrack*.

track
The *Track*.

create_time
When the track was added to the playlist, as seconds since Unix epoch.

creator
The *User* that added the track to the playlist.

is_seen()

set_seen(*value*)

seen
Whether the track is marked as seen or not.

message
A message attached to the track. Typically used in the inbox.

class spotify.**PlaylistType**

class spotify.**PlaylistUnseenTracks**(*session*, *sp_playlistcontainer*, *sp_playlist*)
A list of unseen tracks in a playlist.

The list may contain items that are None.

Returned by *PlaylistContainer.get_unseen_tracks()*.

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

4.1.15 Toplists

class spotify.**Toplist**(*session*, *type=None*, *region=None*, *canonical_username=None*, *callback=None*, *sp_toplistbrowse=None*, *add_ref=True*)
A Spotify toplist of artists, albums or tracks that are currently most popular worldwide or in a specific region.

Call the *get_toplist()* method on your *Session* instance to get a *Toplist* back.

type = None
A *ToplistType* instance that specifies what kind of toplist this is: top artists, top albums, or top tracks.

Changing this field has no effect on existing toplists.

region = None
Either a *ToplistRegion* instance, or a 2-letter ISO 3166-1 country code, that specifies the geographical region this toplist is for.

Changing this field has no effect on existing toplists.

canonical_username = None
If *region* is `ToplistRegion.USER`, then this field specifies which user the toplist is for.

Changing this field has no effect on existing toplists.

loaded_event = None
`threading.Event` that is set when the toplist is loaded.

is_loaded
Whether the toplist's data is loaded yet.

load(*timeout=None*)
Block until the user's data is loaded.

After `timeout` seconds with no results *Timeout* is raised. If `timeout` is `None` the default timeout is used.

The method returns `self` to allow for chaining of calls.

error
An *ErrorType* associated with the toplist.

Check to see if there was problems creating the toplist.

backend_request_duration
The time in ms that was spent waiting for the Spotify backend to create the toplist.

Returns `-1` if the request was served from local cache. Returns `None` if the toplist isn't loaded yet.

tracks
The tracks in the toplist.

Will always return an empty list if the toplist isn't loaded.

albums
The albums in the toplist.

Will always return an empty list if the toplist isn't loaded.

artists
The artists in the toplist.

Will always return an empty list if the toplist isn't loaded.

class spotify.**ToplistRegion**

class spotify.**ToplistType**

4.1.16 Inbox

class spotify.**InboxPostResult**(*session*, *canonical_username=None*, *tracks=None*, *message=''*, *callback=None*, *sp_inbox=None*, *add_ref=True*)
The result object returned by *Session.inbox_post_tracks()*.

loaded_event = None
threading.Event that is set when the inbox post result is loaded.

error
An *ErrorType* associated with the inbox post result.

Check to see if there was problems posting to the inbox.

4.1.17 Social

class spotify.social.**Social**(*session*)
Social sharing controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the social attribute on the Session instance.

private_session
Whether the session is private.

Set to True or False to change.

is_scrobbling(*social_provider*)
Get the ScrobblingState for the given social_provider.

is_scrobbling_possible(*social_provider*)
Check if the scrobbling settings should be shown to the user.

set_scrobbling(*social_provider*, *scrobbling_state*)
Set the scrobbling_state for the given social_provider.

set_social_credentials(*social_provider*, *username*, *password*)
Set the user's credentials with a social provider.

Currently this is only relevant for Last.fm. Call *set_scrobbling()* to force an authentication attempt with the provider. If authentication fails a SCROBBLE_ERROR event will be emitted on the Session object.

class spotify.**ScrobblingState**

class spotify.**SocialProvider**

4.1.18 Player

class spotify.player.**Player**(*session*)
Playback controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the player attribute on the Session instance.

state = 'unloaded'
The player state.

- The state is initially *PlayerState.UNLOADED*.
- When a track is loaded, the state changes to *PlayerState.LOADED*.
- When playback is started the state changes to *PlayerState.PLAYING*.
- When playback is paused the state changes to *PlayerState.PAUSED*.
- When the track is unloaded the state changes to *PlayerState.UNLOADED* again.

load(*track*)
Load `Track` for playback.

seek(*offset*)
Seek to the offset in ms in the currently loaded track.

play(*play=True*)
Play the currently loaded track.

This will cause audio data to be passed to the `music_delivery` callback.

If `play` is set to `False`, playback will be paused.

pause()
Pause the currently loaded track.

This is the same as calling *play()* with `False`.

unload()
Stops the currently playing track.

prefetch(*track*)
Prefetch a `Track` for playback.

This can be used to make libspotify download and cache a track before playing it.

class spotify.player.**PlayerState**

UNLOADED = 'unloaded'

LOADED = 'loaded'

PLAYING = 'playing'

PAUSED = 'paused'

4.1.19 Audio

class spotify.**AudioBufferStats**
Stats about the application's audio buffers.

samples
Number of samples currently in the buffer.

stutter
Number of stutters (audio dropouts) since the last query.

class spotify.**AudioFormat**(*sp_audioformat*)
A Spotify audio format object.

You'll never need to create an instance of this class yourself, but you'll get *AudioFormat* objects as the `audio_format` argument to the `music_delivery` callback.

sample_type
The *SampleType*, currently always `SampleType.INT16_NATIVE_ENDIAN`.

sample_rate
The sample rate, typically 44100 Hz.

channels
The number of audio channels, typically 2.

frame_size()
The byte size of a single frame of this format.

class spotify.**Bitrate**

class spotify.**SampleType**

4.1.20 Audio sinks

class spotify.**AlsaSink**(*session*, *device='default'*)
Audio sink for systems using ALSA, e.g. most Linux systems.

This audio sink requires pyalsaaudio. pyalsaaudio is probably packaged in your Linux distribution.

For example, on Debian/Ubuntu you can install it from APT:

```
sudo apt-get install python-alsaaudio
```

Or, if you want to install pyalsaaudio inside a virtualenv, install the ALSA development headers from APT, then pyalsaaudio:

```
sudo apt-get install libasound2-dev
pip install pyalsaaudio
```

The `device` keyword argument is passed on to `alsaaudio.PCM`. Please refer to the pyalsaaudio documentation for details.

Example:

```
>>> import spotify
>>> session = spotify.Session()
>>> audio = spotify.AlsaSink(session)
>>> loop = spotify.EventLoop(session)
>>> loop.start()
# Login, etc...
>>> track = session.get_track('spotify:track:3N2UhXZI4Gf64Ku3cCjz2g')
>>> track.load()
>>> session.player.load(track)
>>> session.player.play()
# Listen to music...
```

off()
Turn off the audio sink.

This disconnects the sink from the relevant session events.

on()
Turn on the audio sink.

This is done automatically when the sink is instantiated, so you'll only need to call this method if you ever call *off()* and want to turn the sink back on.

class spotify.**PortAudioSink**(*session*)
Audio sink for PortAudio.

PortAudio is available for many platforms, including Linux, OS X, and Windows. This audio sink requires PyAudio. PyAudio is probably packaged in your Linux distribution.

On Debian/Ubuntu you can install PyAudio from APT:

```
sudo apt-get install python-pyaudio
```

Or, if you want to install PyAudio inside a virtualenv, install the PortAudio development headers from APT, then PyAudio:

```
sudo apt-get install portaudio19-dev
pip install --allow-unverified=pyaudio pyaudio
```

On OS X you can install PortAudio using Homebrew:

```
brew install portaudio
pip install --allow-unverified=pyaudio pyaudio
```

For an example of how to use this class, see the *AlsaSink* example. Just replace `AlsaSink` with `PortAudioSink`.

off()
Turn off the audio sink.

This disconnects the sink from the relevant session events.

on()
Turn on the audio sink.

This is done automatically when the sink is instantiated, so you'll only need to call this method if you ever call *off()* and want to turn the sink back on.

4.1.21 Internal API

Warning: This page documents pyspotify's internal APIs. Its intended audience is developers working on pyspotify itself. You should not use anything you find on this page in your own applications.

libspotify CFFI interface

The CFFI wrapper for the full libspotify API is available as *spotify.ffi* and *spotify.lib*.

spotify.**ffi**
`cffi.FFI` instance which knows about libspotify types.

```
>>> import spotify
>>> spotify.ffi.new('sp_audioformat *')
<cdata 'struct sp_audioformat *' owning 12 bytes>
```

spotify.**lib**
Dynamic wrapper around the full libspotify C API.

```
>>> import spotify
>>> msg = spotify.lib.sp_error_message(spotify.lib.SP_ERROR_OK)
>>> msg
<cdata 'char *' 0x7f29fd922cb5>
>>> spotify.ffi.string(msg)
'No error'
```

spotify.lib will always reflect the contents of the `spotify/api.processed.h` file in the pyspotify distribution. To update the API:

1. Update the file `spotify/api.h` with the latest header file from libspotify.
2. Run the Invoke task `preprocess_header` defined in `tasks.py` by running:

```
invoke preprocess_header
```

   The task will update the `spotify/api.processed.h` file.

3. Commit both header files so that they are distributed with pyspotify.

Thread safety utils

spotify.**serialized**(*f*)
Decorator that serializes access to all decorated functions.

The decorator acquires pyspotify's single global lock while calling any wrapped function. It is used to serialize access to:

- All calls to functions on *spotify.lib*.
- All code blocks working on pointers returned from functions on *spotify.lib*.
- All code blocks working on other internal data structures in pyspotify.

Together this is what makes pyspotify safe to use from multiple threads and enables convenient features like the *EventLoop*.

Internal function.

Event emitter utils

class spotify.utils.**EventEmitter**
Mixin for adding event emitter functionality to a class.

on(*event*, *listener*, **user_args*)
Register a `listener` to be called on `event`.

The listener will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()* last.

If the listener function returns `False`, it is removed and will not be called the next time the `event` is emitted.

off(*event=None*, *listener=None*)
Remove a `listener` that was to be called on `event`.

If `listener` is `None`, all listeners for the given `event` will be removed.

If `event` is `None`, all listeners for all events on this object will be removed.

emit(*event*, **event_args*)
Call the registered listeners for `event`.

The listeners will be called with any extra arguments passed to *emit()* first, and then the extra arguments passed to *on()*

num_listeners(*event=None*)
Return the number of listeners for `event`.

Return the total number of listeners for all events on this object if `event` is `None`.

call(*event*, **event_args*)
Call the single registered listener for `event`.

The listener will be called with any extra arguments passed to *call()* first, and then the extra arguments passed to *on()*

Raises `AssertionError` if there is none or multiple listeners for `event`. Returns the listener's return value on success.

Enumeration utils

class spotify.utils.**IntEnum**
An enum type for values mapping to integers.

Tries to stay as close as possible to the enum type specified in **PEP 435** and introduced in Python 3.4.

classmethod add(*name*, *value*)
Add a name-value pair to the enumeration.

spotify.utils.**make_enum**(*lib_prefix*, *enum_prefix=''*)
Class decorator for automatically adding enum values.

The values are read directly from the *spotify.lib* CFFI wrapper around libspotify. All values starting with `lib_prefix` are added. The `lib_prefix` is stripped from the name. Optionally, `enum_prefix` can be specified to add a prefix to all the names.

Object loading utils

spotify.utils.**load**(*session*, *obj*, *timeout=None*)
Block until the object's data is loaded.

If the session isn't logged in, a *spotify.Error* is raised as Spotify objects cannot be loaded without being online and logged in.

The `obj` must at least have the `is_loaded` attribute. If it also has an `error()` method, it will be checked for errors to raise.

After `timeout` seconds with no results *Timeout* is raised.

If unspecified, the `timeout` defaults to 10s. Any timeout is better than no timeout, since no timeout would cause programs to potentially hang forever without any information to help debug the issue.

The method returns `self` to allow for chaining of calls.

Sequence utils

class spotify.utils.**Sequence**(*sp_obj*, *add_ref_func*, *release_func*, *len_func*, *getitem_func*)
Helper class for making sequences from a length and getitem function.

The `sp_obj` is assumed to already have gotten an extra reference through `sp_*_add_ref` and to be automatically released through `sp_*_release` when the `sp_obj` object is GC-ed.

String conversion utils

spotify.utils.**get_with_fixed_buffer**(*buffer_length*, *func*, **args*)
Get a unicode string from a C function that takes a fixed-size buffer.

The C function `func` is called with any arguments given in `args`, a buffer of the given `buffer_length`, and `buffer_length`.

Returns the buffer's value decoded from UTF-8 to a unicode string.

`spotify.utils.`**`get_with_growing_buffer`**(*func*, **args*)
Get a unicode string from a C function that returns the buffer size needed to return the full string.

The C function `func` is called with any arguments given in `args`, a buffer of fixed size, and the buffer size. If the C function returns a size that is larger than the buffer already filled, the C function is called again with a buffer large enough to get the full string from the C function.

Returns the buffer's value decoded from UTF-8 to a unicode string.

`spotify.utils.`**`to_bytes`**(*value*)
Converts bytes, unicode, and C char arrays to bytes.

Unicode strings are encoded to UTF-8.

`spotify.utils.`**`to_bytes_or_none`**(*value*)
Converts C char arrays to bytes and C NULL values to None.

`spotify.utils.`**`to_unicode`**(*value*)
Converts bytes, unicode, and C char arrays to unicode strings.

Bytes and C char arrays are decoded from UTF-8.

`spotify.utils.`**`to_unicode_or_none`**(*value*)
Converts C char arrays to unicode and C NULL values to None.

C char arrays are decoded from UTF-8.

`spotify.utils.`**`to_char`**(*value*)
Converts bytes, unicode, and C char arrays to C char arrays.

`spotify.utils.`**`to_char_or_null`**(*value*)
Converts bytes, unicode, and C char arrays to C char arrays, and `None` to C NULL values.

Country code utils

`spotify.utils.`**`to_country`**(*code*)
Converts a numeric libspotify country code to an ISO 3166-1 two-letter country code in a unicode string.

`spotify.utils.`**`to_country_code`**(*country*)
Converts an ISO 3166-1 two-letter country code in a unicode string to a numeric libspotify country code.

CHAPTER 5

About

5.1 Authors

pyspotify 2.x is copyright 2013-2015 Stein Magnus Jodal and contributors. pyspotify is licensed under the Apache License, Version 2.0.

Thanks to Thomas Adamcik who continuously reviewed code and provided feedback during the development of pyspotify 2.x.

The following persons have contributed to pyspotify 2.x. The list is in the order of first contribution. For details on who have contributed what, please refer to our Git repository.

- Stein Magnus Jodal <stein.magnus@jodal.no>
- Richard Ive <richard@xanox.net>
- Thomas Vander Stichele <thomas@apestaart.org>
- Nick Steel <kingosticks@gmail.com>
- Trygve Aaberge <trygveaa@gmail.com>
- Tarik Dadi <daditarik@gmail.com>
- vrs01 <vrs01@users.noreply.github.com>
- Thomas Adamcik <thomas@adamcik.no>
- Edward Betts <edward@4angle.com>

If you want to contribute to pyspotify, see *Contributing*.

5.2 Changelog

5.2.1 v2.1.1 (2019-11-17)

Maintanance release.

- Add support for Python 3.8. No changes was required, but the test suite now runs on this version too.
- Switch from Travis CI to CircleCI.

5.2.2 v2.1.0 (2019-07-08)

Maintenance release.

- Drop support for Python 3.3 and 3.4, as both has reached end of life.
- Add support for Python 3.6 and 3.7. No changes was required, but the test suite now runs on these versions too.
- On Python 3, import `Iterable`, `MutableSequence`, and `Sequence` from `collections.abc` instead of `collections`. This fixes a deprecation warning on Python 3.7 and prepares for Python 3.8.
- Document that the search API is broken. If it is used, raise an exception instead of sending the search to Spotify, as that seems to disconnect your session. (Fixes: #183)
- Format source code with Black.

5.2.3 v2.0.5 (2015-09-22)

Bug fix release.

- To follow up on the previous release, the getters for the proxy configs now convert empty strings in the `sp_session_config` struct back to `None`. Thus, the need to set these configs to empty strings in the struct to make sure the cached settings are cleared from disk are now an internal detail, hidden from the user of pyspotify.
- Make *`tracefile`* default to `None` and set to `NULL` in the libspotify config struct. If it is set to an empty string by default, libspotify will try to use a file with an empty filename for cache and fail with "LibError: Unable to open trace file". Now empty strings are set as `NULL` in the `sp_session_config` struct. (Fixes: mopidy-spotify#70)
- libspotify segfaults if the `device_id` config is set to an empty string. We now avoid this segfault if *`device_id`* is set to an empty string by setting the `device_id` field in libspotify's `sp_session_config` struct to `NULL` instead.
- As some test tools (like coverage.py 4.0) no longer support Python 3.2, we no longer test pyspotify on Python 3.2. Though, we have not done anything to intentionally break support for Python 3.2 ourselves.

5.2.4 v2.0.4 (2015-09-15)

Bug fix release.

- It has been observed that libspotify will reuse cached proxy settings from previous sessions if the proxy fields on the `sp_session_config` struct are set to `NULL`. When the `sp_session_config` fields are set to an empty string, the cached settings are updated. When attributes on *`spotify.Config`* are set to `None`, we now set the fields on `sp_session_config` to empty strings instead of `NULL`.

5.2.5 v2.0.3 (2015-09-05)

Bug fix release.

- Make moving a playlist to its own location a no-op instead of causing an error like libspotify does. (Fixes: #175)
- New better installation instructions. (Fixes: #174)

5.2.6 v2.0.2 (2015-08-06)

Bug fix release.

- Use `sp_session_starred_for_user_create(session, username)` instead of `sp_playlist_create(session, link)` to get starred playlists by URI. The previous approach caused segfaults under some circumstances. (Fixes: mopidy-spotify#60)

5.2.7 v2.0.1 (2015-07-20)

Bug fix release.

- Make *`spotify.Session.get_playlist()`* acquire the global lock before modifying the global playlist cache.
- Make *`Playlist`* and *`PlaylistContainer`* register callbacks with libspotify if and only if a Python event handler is added to the object. Previously, we always registered the callbacks with libspotify. Hopefully, this will remove the preconditions for the crashes in #122, #153, and #165.

5.2.8 v2.0.0 (2015-06-01)

pyspotify 2.x is a full rewrite of pyspotify. While pyspotify 1.x is a CPython C extension, pyspotify 2.x uses CFFI to wrap the libspotify C library. It works on CPython 2.7 and 3.2+, as well as PyPy 2.6+. pyspotify 2.0 makes 100% of the libspotify 12.1.51 API available from Python, going far beyond the API coverage of pyspotify 1.x.

The following are the changes since pyspotify 2.0.0b5.

Dependency changes

- Require cffi >= 1.0. (Fixes: #133, #160)
- If you're using pyspotify with PyPy you need version 2.6 or newer as older versions of PyPy come with a too old cffi version. For PyPy3, you'll probably need the yet to be released PyPy3 2.5.

ALSA sink

- Changed the *`spotify.AlsaSink`* keyword argument `card` to `device` to align with pyalsaaudio 0.8.
- Updated to work with pyalsaaudio 0.8 which changed the signature of `alsaaudio.PCM`. *`spotify.AlsaSink`* still works with pyalsaaudio 0.7, but 0.8 is recommended at least for Python 3 users, as it fixes a memory leak present on Python 3 (see #127). (Fixes: #162)

5.2.9 v2.0.0b5 (2015-05-09)

A fifth beta with a couple of bug fixes.

Minor changes

- Changed *spotify.Link.as_playlist()* to also support creating playlists from links with type spotify.LinkType.STARRED.
- Changed all load() methods to raise *spotify.Error* instead of RuntimeError if the session isn't logged in.
- Changed from nose to py.test as test runner.

Bug fixes

- Work around segfault in libspotify when *spotify.Config.cache_location* is set to None and then used to create a session. (Fixes: #151)
- Return a *spotify.PlaylistPlaceholder* object instead of raising an exception if the playlist container contains an element of type PLACEHOLDER. (Fixes: #159)

5.2.10 v2.0.0b4 (2015-01-13)

The fourth beta includes a single API change, a couple of API additions, and otherwise minor tweaks to logging.

pyspotify 2.x has been verified to work on PyPy3, and PyPy3 is now part of the test matrix.

Minor changes

- Added *spotify.Link.url* which returns an https://open.spotify.com/... URL for the link object.
- Adjusted info, warning, and error level log messages to include the word "Spotify" or "pyspotify" for context in applications not including the logger name in the log. debug level messages have not been changed, as it is assumed that more details, including the logger name, is included in debug logs.
- Added *spotify.player.Player.state* which is maintained by calls to the various *Player* methods.

Bug fixes

- Fix *spotify.Playlist.reorder_tracks()*. It now accepts a list of track indexes instead of a list of tracks. This makes it possible to reorder any of multiple identical tracks in a playlist and is consistent with *spotify.Playlist.remove_tracks()*. (Fixes: #134)
- Fix pause/resume/stop in the examples/shell.py example. (PR: #140)
- Errors passed to session callbacks are now logged with the full error type representation, instead of just the integer value. E.g. where previously only "8" was logged, we now log "<ErrorType.UNABLE_TO_CONTACT_SERVER: 8>".

5.2.11 v2.0.0b3 (2014-05-04)

The third beta includes a couple of changes to the API in the name of consistency, as well as three minor improvements.

Also worth noticing is that with this release, pyspotify 2.x has been in development for a year and a day. Happy birthday, pyspotify 2!

Refactoring: Connection cleanup

Parts of *spotify.Session* and *spotify.Session.offline* has been moved to *spotify.Session.connection*:

- `set_connection_type()` has been replaced by *session.connection.type*, which now also allows reading the current connection type.
- `set_connection_rules()` has been replaced by:
  - *allow_network*
  - *allow_network_if_roaming*
  - *allow_sync_over_wifi*
  - *allow_sync_over_mobile*

  The new attributes allow reading the current connection rules, so your application don't have to keep track of what rules it has set.
- `session.connection_state` has been replaced by *session.connection.state*

Refactoring: position vs index

Originally, pyspotify named everything identically with libspotify and have thus ended up with a mix of the terms "position" and "index" for the same concept. Now, we use "index" all over the place, as that's also the name used in the Python world at large. This changes the signature of three methods, which may affect you if you use keyword arguments to call the methods. There's also a number of affected events, but these changes shouldn't stop your code from working.

Affected functions include:

- *spotify.Playlist.add_tracks()* now takes `index` instead of `position`.
- *spotify.Playlist.remove_tracks()* now takes `indexes` instead of `positions`.
- *spotify.Playlist.reorder_tracks()* now takes `new_index` instead of `new_position`.

Affected events include:

- *spotify.PlaylistContainerEvent.PLAYLIST_ADDED*
- *spotify.PlaylistContainerEvent.PLAYLIST_REMOVED*
- *spotify.PlaylistContainerEvent.PLAYLIST_MOVED*
- *spotify.PlaylistEvent.TRACKS_ADDED*
- *spotify.PlaylistEvent.TRACKS_REMOVED*
- *spotify.PlaylistEvent.TRACKS_MOVED*
- *spotify.PlaylistEvent.TRACK_CREATED_CHANGED*
- *spotify.PlaylistEvent.TRACK_SEEN_CHANGED*
- *spotify.PlaylistEvent.TRACK_MESSAGE_CHANGED*

Minor changes

- `load()` methods now return the object if it is already loaded, even if *state* isn't `LOGGED_IN`. Previously, a `RuntimeError` was raised requiring the session to be logged in and online before loading already loaded objects.

- *spotify.Playlist.tracks* now implements the `collections.MutableSequence` contract, supporting deleting items with `del playlist.tracks[i]`, adding items with `playlist.tracks[i] = track`, etc.
- *spotify.Session.get_link()* and all other methods accepting Spotify URIs now also understand open.spotify.com and play.spotify.com URLs.

5.2.12 v2.0.0b2 (2014-04-29)

The second beta is a minor bug fix release.

Bug fixes

- Fix *spotify.Playlist.remove_tracks*. It now accepts a list of track positions instead of a list of tracks. This makes it possible to remove any of multiple identical tracks in a playlist. (Fixes: #128)

Minor changes

- Make all objects compare as equal and have the same hash if they wrap the same libspotify object. This makes it possible to find the index of a track in a playlist by doing `playlist.tracks.index(track)`, where `playlist.tracks` is a custom collection always returning new *Track* instances. (Related to: #128)
- *spotify.Config.ca_certs_filename* now works on systems where libspotify has this field. On systems where this field isn't present in libspotify, assigning to it will have no effect. Previously, assignment to this field was a noop on all platforms because the field is missing from libspotify on OS X.

5.2.13 v2.0.0b1 (2014-04-24)

pyspotify 2.x is a full rewrite of pyspotify. While pyspotify 1.x is a CPython C extension, pyspotify 2.x uses CFFI to make 100% of the libspotify C library available from Python. It works on CPython 2.7 and 3.2+, as well as PyPy 2.1+.

Since the previous release, pyspotify has become thread safe. That is, pyspotify can safely be used from multiple threads. The added thread safety made an integrated event loop possible, which greatly simplifies the usage of pyspotify, as can be seen from the updated example in `examples/shell.py`. Audio sink helpers for ALSA and PortAudio have been added, together with updated examples that can play music. A number of bugs have been fixed, and at the time of the release, there are no known issues.

The pyspotify 2.0.0b1 release marks the completion of all planned features for pyspotify 2.x. The plans for the next releases are focused on fixing bugs as they surface, incrementally improving the documentation, and integrating feedback from increased usage of the library in the wild.

Feature: Thread safety

- Hold the global lock while we are working with pointers returned by libspotify. This ensures that we never call libspotify from another thread while we are still working on the data returned by the previous libspotify call, which could make the data garbage.
- Ensure we never edit shared data structures without holding the global lock.

Feature: Event loop

- Add *spotify.EventLoop* helper thread that reacts to *NOTIFY_MAIN_THREAD* events and calls *process_events()* for you when appropriate.
- Update `examples/shell.py` to be a lot simpler with the help of the new event loop.

Feature: Audio playback

- Add *spotify.AlsaSink*, an audio sink for playback through ALSA on Linux systems.
- Add *spotify.PortAudioSink*, an audio sink for playback through PortAudio on most platforms, including Linux, OS X, and Windows.
- Update `examples/shell.py` to use the ALSA sink to play music.
- Add `examples/play_track.py` as a simpler example of audio playback.

Refactoring: Remove global state

To prepare for removing all global state, the use of the module attribute `spotify.session_instance` has been replaced with explicit passing of the session object to all objects that needs it. To allow for this, the following new methods have been added, and should be used instead of their old equivalents:

- *spotify.Session.get_link()* replaces *spotify.Link*.
- *spotify.Session.get_track()* replaces *spotify.Track*.
- *spotify.Session.get_local_track()* replaces `spotify.LocalTrack`.
- *spotify.Session.get_album()* replaces *spotify.Album*.
- *spotify.Session.get_artist()* replaces *spotify.Artist*.
- *spotify.Session.get_playlist()* replaces *spotify.Playlist*.
- *spotify.Session.get_user()* replaces *spotify.User*.
- *spotify.Session.get_image()* replaces *spotify.Image*.
- *spotify.Session.get_toplist()* replaces *spotify.Toplist*.

Refactoring: Consistent naming of `Session` members

With all the above getters added to the *spotify.Session* object, it made sense to rename some existing methods of *Session* for consistency:

- `spotify.Session.starred_for_user()` is replaced by *get_starred()*.
- `spotify.Session.starred` to get the currently logged in user's starred playlist is replaced by *get_starred()* without any arguments.
- *spotify.Session.get_published_playlists()* replaces `published_playlists_for_user()`. As previously, it returns the published playlists for the currently logged in user if no username is provided.

Refactoring: Consistent naming of `threading.Event` objects

All `threading.Event` objects have been renamed to be consistently named across classes.

- *spotify.AlbumBrowser.loaded_event* replaces `spotify.AlbumBrowser.complete_event`.
- *spotify.ArtistBrowser.loaded_event* replaces `spotify.ArtistBrowser.complete_event`.
- *spotify.Image.loaded_event* replaces `spotify.Image.load_event`.
- *spotify.InboxPostResult.loaded_event* replaces `spotify.InboxPostResult.complete_event`.
- *spotify.Search.loaded_event* replaces `spotify.Search.complete_event`.
- *spotify.Toplist.loaded_event* replaces `spotify.Toplist.complete_event`.

Refactoring: Change how to register image load listeners

pyspotify has two main schemes for registering listener functions:

- Objects that only emit an event when it is done loading, like *AlbumBrowser*, *ArtistBrowser*, *InboxPostResult*, *Search*, and *Toplist*, accept a single callback as a `callback` argument to its constructor or constructor methods.
- Objects that have multiple callback events, like *Session*, *PlaylistContainer*, and *Playlist*, accept the registration and unregistration of one or more listener functions for each event it emits. This can happen any time during the object's life cycle.

Due to pyspotify's close mapping to libspotify's organization, `Image` objects used to use a third variant with two methods, `add_load_callback()` and `remove_load_callback()`, for adding and removing load callbacks. These methods have now been removed, and *Image* accepts a `callback` argument to its constructor and constructor methods:

- *spotify.Album.cover()* accepts a `callback` argument.
- *spotify.Artist.portrait()* accepts a `callback` argument.
- *spotify.ArtistBrowser.portraits()* is now a method and accepts a `callback` argument.
- *spotify.Link.as_image()* accepts a `callback` argument.
- *spotify.Playlist.image()* is now a method and accepts a `callback` argument.
- *spotify.Session.get_image()* accepts a `callback` argument.

Bug fixes

- Remove multiple extra `sp_link_add_ref()` calls, potentially causing memory leaks in libspotify.
- Add missing error check to *spotify.Playlist.add_tracks()*.
- Keep album, artist, image, inbox, search, and toplist objects alive until their complete/load callbacks have been called, even if the library user doesn't keep any references to the objects. (Fixes: #121)
- Fix flipped logic causing crash in *spotify.Album.cover_link()*. (Fixes: #126)
- Work around segfault in libspotify if *private_session* is set before the session is logged in and the first events are processed. This is a bug in libspotify which has been reported to Spotify through their IRC channel.

- Multiple attributes on *Track* raised an exception if accessed before the track was loaded. They now return None or similar as documented.
- Fix segfault when creating local tracks without all arguments specified. NULL was used as the placeholder instead of the empty string.
- Support negative indexes on all custom sequence types. For example, collection[-1] returns the last element in the collection.
- We now cache playlists when created from URIs. Previously, only playlists created from sp_playlist objects were cached. This avoids a potentially large number of wrapper object recreations due to a flood of updates to the playlist when it is initially loaded. Combined with having registered a callback for the libspotify playlist_update_in_progress callback, this could cause deep call stacks reaching the maximum recursion depth. (Fixes: #122)

Minor changes

- Add *spotify.get_libspotify_api_version()* and *spotify.get_libspotify_build_id()*.
- Running python setup.py test now runs the test suite.
- The tests are now compatible with CPython 3.4. No changes to the implementation was required.
- The test suite now runs on Mac OS X, using CPython 2.7, 3.2, 3.3, 3.4, and PyPy 2.2, on every push to GitHub.

5.2.14 v2.0.0a1 (2014-02-14)

pyspotify 2.x is a full rewrite of pyspotify. While pyspotify 1.x is a CPython C extension, pyspotify 2.x uses CFFI to wrap the libspotify C library. It works on CPython 2.7 and 3.2+, as well as PyPy 2.1+.

This first alpha release of pyspotify 2.0.0 makes 100% of the libspotify 12.1.51 API available from Python, going far beyond the API coverage of pyspotify 1.x.

pyspotify 2.0.0a1 has an extensive test suite with 98% line coverage. All tests pass on all combinations of CPython 2.7, 3.2, 3.3, PyPy 2.2 running on Linux on i386, amd64, armel, and armhf. Mac OS X should work, but has not been tested recently.

This release *does not* provide:

- thread safety,
- an event loop for regularly processing libspotify events, or
- audio playback drivers.

These features are planned for the upcoming prereleases.

Development milestones

- 2014-02-13: Playlist callbacks complete. pyspotify 2.x now covers 100% of the libspotify 12 API. Docs reviewed, quickstart guide extended. Redundant getters/setters removed.
- 2014-02-08: Playlist container callbacks complete.
- 2014-01-31: Redesign session event listening to a model supporting multiple listeners per event, with a nicer API for registering listeners.
- 2013-12-16: Ensure we never call libspotify from two different threads at the same time. We can't assume that the CPython GIL will ensure this for us, as we target non-CPython interpreters like PyPy.

- 2013-12-13: Artist browsing complete.
- 2013-12-13: Album browsing complete.
- 2013-11-29: Toplist subsystem complete.
- 2013-11-27: Inbox subsystem complete.
- 2013-10-14: Playlist subsystem *almost* complete.
- 2013-06-21: Search subsystem complete.
- 2013-06-10: Album subsystem complete.
- 2013-06-09: Track and artist subsystem complete.
- 2013-06-02: Session subsystem complete, with all methods.
- 2013-06-01: Session callbacks complete.
- 2013-05-25: Session config complete.
- 2013-05-16: Link subsystem complete.
- 2013-05-09: User subsystem complete.
- 2013-05-08: Session configuration and creation, with login and logout works.
- 2013-05-03: The Python object `spotify.lib` is a working CFFI wrapper around the entire libspotify 12 API. This will be the foundation for more pythonic APIs. The library currently works on CPython 2.7, 3.3 and PyPy 2.

5.2.15 v1.x series

See the pyspotify 1.x changelog.

5.3 Contributing

Contributions to pyspotify are welcome! Here are some tips to get you started hacking on pyspotify and contributing back your patches.

5.3.1 Development setup

1. Make sure you have the following Python versions installed:
   - CPython 2.7
   - CPython 3.5
   - CPython 3.6
   - CPython 3.7
   - PyPy2.7 6.0+
   - PyPy3.5 6.0+

   If you're on Ubuntu, the Dead Snakes PPA has packages of both old and new Python versions.

2. Install the following with development headers: Python, libffi, and libspotify.

   On Debian/Ubuntu, make sure you have apt.mopidy.com in your APT sources to get the libspotify package, then run:

```
sudo apt-get install python-all-dev python3-all-dev libffi-dev libspotify-dev
```

3. Create and activate a virtualenv:

```
virtualenv ve
source ve/bin/activate
```

4. Install development dependencies:

```
pip install -e ".[dev]
```

5. Run tests.

   For a quick test suite run, using the virtualenv's Python version:

```
py.test
```

   For a complete test suite run, using all the Python implementations:

```
tox
```

6. For some more development task helpers, install `invoke`:

```
pip install invoke
```

   To list available tasks, run:

```
invoke --list
```

   For example, to run tests on any file change, run:

```
invoke test --watch
```

   Or, to build docs when any file changes, run:

```
invoke docs --watch
```

   See the file `tasks.py` for the task definitions.

5.3.2 Submitting changes

- Code should be accompanied by tests and documentation. Maintain our excellent test coverage.
- Follow the existing code style, especially make sure `flake8` does not complain about anything.
- Write good commit messages. Here's three blog posts on how to do it right:
  - Writing Git commit messages
  - A Note About Git Commit Messages
  - On commit messages
- One branch per feature or fix. Keep branches small and on topic.
- Send a pull request to the `v2.x/master` branch. See the GitHub pull request docs for help.

5.3.3 Additional resources

- Issue tracker
- Mailing List
- GitHub documentation
- libspotify documentation

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y