python-sounddevice

Release 0.3.2

Matthias Geier

March 16, 2016

Contents

This Python[1] module provides bindings for the PortAudio[2] library and a few convenience functions to play and record NumPy[3] arrays containing audio signals.

Documentation: http://python-sounddevice.rtfd.org/

Code: http://github.com/spatialaudio/python-sounddevice/

Python Package Index: http://pypi.python.org/pypi/sounddevice/

License: MIT – see the file `LICENSE` for details.

1 Requirements

Python: Of course, you'll need Python[4]. Any version where CFFI (see below) is supported should work. If you don't have Python installed yet, you should get one of the distributions which already include CFFI and

[1] http://www.python.org/
[2] http://www.portaudio.com/
[3] http://www.numpy.org/
[4] http://www.python.org/

NumPy (and many other useful things), e.g. Anaconda[5] or WinPython[6].

pip/setuptools: Those are needed for the installation of the Python module and its dependencies. Most systems will have these installed already, but if not, you should install it with your package manager or you can download and install pip and setuptools as described on the pip installation[7] page. If you happen to have pip but not setuptools, use this command:

```
pip install setuptools --user
```

CFFI: The C Foreign Function Interface for Python[8] is used to access the C-API of the PortAudio library from within Python. It supports CPython 2.6, 2.7, 3.x; and is distributed with PyPy[9] 2.0 beta2 or later. If it's not installed already, you should install it with your package manager (the package might be called `python3-cffi`, `python-cffi` or similar), or you can get it with:

```
pip install cffi --user
```

PortAudio library: The PortAudio[10] library must be installed on your system (and CFFI must be able to find it). Again, you should use your package manager to install it (the package might be called `libportaudio2` or similar). If you prefer, you can of course also download the sources and compile the library yourself. If you are using Mac OS X or Windows, the library will be installed automagically with *pip* (see "Installation" below).

NumPy (optional): NumPy[11] is only needed if you want to play back and record NumPy arrays. The classes *`sounddevice.RawStream`*, *`sounddevice.RawInputStream`* and *`sounddevice.RawOutputStream`* use plain Python buffer objects and don't need NumPy at all. If you need NumPy, you should install it with your package manager or use a Python distribution that already includes NumPy (see above). Installing NumPy with pip is not recommended.

2 Installation

Once you have installed the above-mentioned dependencies, you can use pip to download and install the latest release with a single command:

```
pip install sounddevice --user
```

If you want to install it system-wide for all users (assuming you have the necessary rights), you can just drop the `--user` option.

To un-install, use:

```
pip uninstall sounddevice
```

3 Usage

First, import the module:

```
import sounddevice as sd
```

3.1 Playback

Assuming you have a NumPy array named `myarray` holding audio data with a sampling frequency of `fs` (in the most cases this will be 44100 or 48000 frames per second), you can play it back with *`sounddevice.play()`*:

[5] http://docs.continuum.io/anaconda/
[6] http://winpython.github.io/
[7] http://www.pip-installer.org/en/latest/installing.html
[8] http://cffi.readthedocs.org/
[9] http://pypy.org/
[10] http://www.portaudio.com/
[11] http://www.numpy.org/

```
sd.play(myarray, fs)
```

This function returns immediately but continues playing the audio signal in the background. You can stop playback with *sounddevice.stop()*:

```
sd.stop()
```

If you know that you will use the same sampling frequency for a while, you can set it as default using *sounddevice.default.samplerate*:

```
sd.default.samplerate = fs
```

After that, you can drop the *samplerate* argument:

```
sd.play(myarray)
```

3.2 Recording

To record audio data from your sound device into a NumPy array, use *sounddevice.rec()*:

```
duration = 10  # seconds
myrecording = sd.rec(duration * fs, samplerate=fs, channels=2)
```

Again, for repeated use you can set defaults using *sounddevice.default*:

```
sd.default.samplerate = fs
sd.default.channels = 2
```

After that, you can drop the additional arguments:

```
myrecording = sd.rec(duration * fs)
```

This function also returns immediately but continues recording in the background. In the meantime, you can run other commands. If you want to check if the recording is finished, you should use *sounddevice.wait()*:

```
sd.wait()
```

If the recording was already finished, this returns immediately; if not, it waits and returns as soon as the recording is finished.

Alternatively, you could have used the *blocking* argument in the first place:

```
myrecording = sd.rec(duration * fs, blocking=True)
```

By default, the recorded array has the data type `'float32'` (see *sounddevice.default.dtype*), but this can be changed with the *dtype* argument:

```
myrecording = sd.rec(duration * fs, dtype='float64')
```

3.3 Simultaneous Playback and Recording

To play back an array and record at the same time, use *sounddevice.playrec()*:

```
myrecording = sd.playrec(myarray, fs, channels=2)
```

The number of output channels is obtained from `myarray`, but the number of input channels still has to be specified.

Again, default values can be used:

```
sd.default.samplerate = fs
sd.default.channels = 2
myrecording = sd.playrec(myarray)
```

In this case the number of output channels is still taken from `myarray` (which may or may not have 2 channels), but the number of input channels is taken from *sounddevice.default.channels*.

3.4 Device Selection

In many cases, the default input/output device(s) will be the one(s) you want, but it is of course possible to choose a different device. Use *sounddevice.query_devices()* to get a list of supported devices. The same list can be obtained from a terminal by typing the command

```
python -m sounddevice
```

You can use the corresponding device ID to select a desired device by assigning to *sounddevice.default.device* or by passing it as *device* argument to *sounddevice.play()*, *sounddevice.Stream()* etc.

Instead of the numerical device ID, you can also use a space-separated list of case-insensitive substrings of the device name (and the host API name, if needed). See *sounddevice.default.device* for details.

```
import sounddevice as sd
sd.default.samplerate = 44100
sd.default.device = 'digital output'
sd.play(myarray)
```

3.5 Callback Streams

Callback "wire" with *sounddevice.Stream*:

```
import sounddevice as sd
duration = 5  # seconds

def callback(indata, outdata, frames, time, status):
    if status:
        print(status, flush=True)
    outdata[:] = indata

with sd.Stream(channels=2, callback=callback):
    sd.sleep(duration * 1000)
```

Same thing with *sounddevice.RawStream*:

```
import sounddevice as sd
duration = 5  # seconds

def callback(indata, outdata, frames, time, status):
    if status:
        print(status, flush=True)
    outdata[:] = indata

with sd.RawStream(channels=2, dtype='int24', callback=callback):
    sd.sleep(duration * 1000)
```

Note: We are using 24-bit samples here for no particular reason (just because we can).

3.6 Blocking Read/Write Streams

Instead of using a callback function, you can also use the blocking methods *sounddevice.Stream.read()* and *sounddevice.Stream.write()* (and of course the corresponding methods in

sounddevice.InputStream, *sounddevice.OutputStream*, *sounddevice.RawStream*, *sounddevice.RawInputStream* and *sounddevice.RawOutputStream*).

4 Contributing

If you find bugs, errors, omissions or other things that need improvement, please create an issue or a pull request at http://github.com/spatialaudio/python-sounddevice/. Contributions are always welcome!

Instead of pip-installing the latest release from PyPI, you should get the newest development version from Github[12]:

```
git clone --recursive https://github.com/spatialaudio/python-sounddevice.git
cd python-sounddevice
python setup.py develop --user
```

This way, your installation always stays up-to-date, even if you pull new changes from the Github repository.

If you prefer, you can also replace the last command with:

```
pip install --user -e .
```

... where `-e` stands for `--editable`.

If you used the `--recursive` option when cloning, the dynamic libraries for Mac OS X and Windows should be available. If not, you can get the submodule with:

```
git submodule update --init --recursive
```

If you make changes to the documentation, you can re-create the HTML pages using Sphinx[13]. You can install it and a few other necessary packages with:

```
pip install -r doc/requirements.txt --user
```

To create the HTML pages, use:

```
python setup.py build_sphinx
```

The generated files will be available in the directory `build/sphinx/html/`.

5 API Documentation

Play and Record Sound with Python.

http://python-sounddevice.rtfd.org/

`sounddevice.`**`play`**(*data*, *samplerate=None*, *mapping=None*, *blocking=False*, ***kwargs*)

Play back an array of audio data.

Parameters

- **data** (*array_like*) – Audio data to be played back. The columns of a two-dimensional array are interpreted as channels, one-dimensional arrays are treated as mono data. The data types *float64*, *float32*, *int32*, *int16*, *int8* and *uint8* can be used. *float64* data is converted to *float32* before passing it to PortAudio, because it's not supported natively.
- **mapping** (*array_like, optional*) – List of channel numbers (starting with 1) where the columns of *data* shall be played back on. Must have the same length as number of channels in *data* (except if *data* is mono). Each channel may only appear once in *mapping*.

[12] http://github.com/spatialaudio/python-sounddevice/
[13] http://sphinx-doc.org/

- **blocking** (*bool, optional*) – If `False` (the default), return immediately (but playback continues in the background), if `True`, wait until playback is finished. A non-blocking invocation can be stopped with *stop()* or turned into a blocking one with *wait()*.

Other Parameters **samplerate, **kwargs** – All parameters of *OutputStream* (except *channels*, *dtype*, *callback* and *finished_callback*) can be used.

See also:

rec(), *playrec()*

sounddevice.**rec**(*frames=None*, *samplerate=None*, *channels=None*, *dtype=None*, *out=None*, *mapping=None*, *blocking=False*, ***kwargs*)

Record audio data.

Parameters

- **frames** (*int, sometimes optional*) – Number of frames to record. Not needed if *out* is given.
- **channels** (*int, optional*) – Number of channels to record. Not needed if *mapping* or *out* is given. The default value can be changed with *default.channels*.
- **dtype** (*str or numpy.dtype, optional*) – Data type of the recording. Not needed if *out* is given. The data types *float64*, *float32*, *int32*, *int16*, *int8* and *uint8* can be used. For *dtype='float64'*, audio data is recorded in *float32* format and converted afterwards, because it's not natively supported by PortAudio. The default value can be changed with *default.dtype*.
- **mapping** (*array_like, optional*) – List of channel numbers (starting with 1) to record. If *mapping* is given, *channels* is silently ignored.
- **blocking** (*bool, optional*) – If `False` (the default), return immediately (but recording continues in the background), if `True`, wait until recording is finished. A non-blocking invocation can be stopped with *stop()* or turned into a blocking one with *wait()*.

Returns

numpy.ndarray or type(out) – The recorded data.

Note: By default (`blocking=False`), an array of data is returned which is still being written to while recording. The returned data is only valid once recording has stopped. Use *wait()* to make sure the recording is finished.

Other Parameters

- **out** (*numpy.ndarray or subclass, optional*) – If *out* is specified, the recorded data is written into the given array instead of creating a new array. In this case, the arguments *frames*, *channels* and *dtype* are silently ignored! If *mapping* is given, its length must match the number of channels in *out*.
- **samplerate, **kwargs** – All parameters of *InputStream* (except *callback* and *finished_callback*) can be used.

See also:

play(), *playrec()*

sounddevice.**playrec**(*data*, *samplerate=None*, *channels=None*, *dtype=None*, *out=None*, *input_mapping=None*, *output_mapping=None*, *blocking=False*, ***kwargs*)

Simultaneous playback and recording.

Parameters

- **data** (*array_like*) – Audio data to be played back. See *play()*.

- **channels** (*int, sometimes optional*) – Number of input channels, see *rec()*. The number of output channels is obtained from *data.shape*.
- **dtype** (*str or numpy.dtype, optional*) – Input data type, see *rec()*. If *dtype* is not specified, it is taken from *data.dtype* (i.e. *default.dtype* is ignored). The output data type is obtained from *data.dtype* anyway.
- **input_mapping, output_mapping** (*array_like, optional*) – See the parameter *mapping* of *rec()* and *play()*, respectively.
- **blocking** (*bool, optional*) – If `False` (the default), return immediately (but continue playback/recording in the background), if `True`, wait until playback/recording is finished. A non-blocking invocation can be stopped with *stop()* or turned into a blocking one with *wait()*.

Returns *numpy.ndarray or type(out)* – The recorded data. See *rec()*.

Other Parameters

- **out** (*numpy.ndarray or subclass, optional*) – See *rec()*.
- **samplerate, **kwargs** – All parameters of *Stream* (except *channels*, *dtype*, *callback* and *finished_callback*) can be used.

See also:

play(), *rec()*

sounddevice.**wait**()

Wait for *play()*/*rec()*/*playrec()* to be finished.

Playback/recording can be stopped with a `KeyboardInterrupt`[14].

Returns *CallbackFlags or None* – If at least one buffer over-/underrun happened during the last playback/recording, a *CallbackFlags* object is returned.

See also:

get_status()

sounddevice.**stop**(*ignore_errors=True*)

Stop playback/recording.

This only stops *play()*, *rec()* and *playrec()*, but has no influence on streams created with *Stream*, *InputStream*, *OutputStream*, *RawStream*, *RawInputStream*, *RawOutputStream*.

sounddevice.**get_status**()

Get information about over-/underflows in play()/rec()/playrec().

Returns *CallbackFlags* – A *CallbackFlags* object that holds information about the last invocation of *play()*, *rec()* or *playrec()*.

See also:

wait()

sounddevice.**query_devices**(*device=None*, *kind=None*)

Return information about available devices.

Information and capabilities of PortAudio devices. Devices may support input, output or both input and output.

To find the default input/output device, use *default.device*.

Parameters

- **device** (*int or str, optional*) – Numeric device ID or device name substring(s). If specified, information about only the given *device* is returned in a single dictionary.

[14] http://docs.python.org/3/library/exceptions.html#KeyboardInterrupt

- **kind** (*{'input', 'output'}, optional*) – If *device* is not specified and *kind* is `'input'` or `'output'`, a single dictionary is returned with information about the default input or output device, respectively.

Returns

dict or DeviceList – A dictionary with information about the given *device* or – if no *device* was specified – a *DeviceList* containing one dictionary for each available device. The dictionaries have the following keys:

`'name'` The name of the device.

`'hostapi'` The ID of the corresponding host API. Use *query_hostapis()* to get information about a host API.

`'max_input_channels'`,`'max_output_channels'` The maximum number of input/output channels supported by the device. See *default.channels*.

`'default_low_input_latency'`,`'default_low_output_latency'` Default latency values for interactive performance. This is used if *default.latency* (or the *latency* argument of *playrec()*, *Stream* etc.) is set to `'low'`.

`'default_high_input_latency'`,`'default_high_output_latency'` Default latency values for robust non-interactive applications (e.g. playing sound files). This is used if *default.latency* (or the *latency* argument of *playrec()*, *Stream* etc.) is set to `'high'`.

`'default_samplerate'` The default sampling frequency of the device. This is used if *default.samplerate* is not set.

Notes

The list of devices can also be displayed in a terminal:

```
python -m sounddevice
```

Examples

The returned *DeviceList* can be indexed and iterated over like a normal `tuple`[15] (yielding the above-mentioned dictionaries), but it also has a special string representation which is shown when used in an interactive Python session.

Each available device is listed on one line together with the corresponding device ID, which can be assigned to *default.device* or used as *device* argument in *play()*, *Stream* etc.

The first character of a line is > for the default input device, < for the default output device and * for the default input/output device. After the device ID and the device name, the corresponding host API name is displayed. In the end of each line, the maximum number of input and output channels is shown.

On a GNU/Linux computer it might look somewhat like this:

```
>>> import sounddevice as sd
>>> sd.query_devices()
   0 HDA Intel: ALC662 rev1 Analog (hw:0,0), ALSA (2 in, 2 out)
   1 HDA Intel: ALC662 rev1 Digital (hw:0,1), ALSA (0 in, 2 out)
   2 HDA Intel: HDMI 0 (hw:0,3), ALSA (0 in, 8 out)
   3 sysdefault, ALSA (128 in, 128 out)
   4 front, ALSA (0 in, 2 out)
   5 surround40, ALSA (0 in, 2 out)
   6 surround51, ALSA (0 in, 2 out)
```

[15] http://docs.python.org/3/library/stdtypes.html#tuple

```
       7 surround71, ALSA (0 in, 2 out)
       8 iec958, ALSA (0 in, 2 out)
       9 spdif, ALSA (0 in, 2 out)
      10 hdmi, ALSA (0 in, 8 out)
    * 11 default, ALSA (128 in, 128 out)
      12 dmix, ALSA (0 in, 2 out)
      13 /dev/dsp, OSS (16 in, 16 out)
```

Note that ALSA provides access to some "real" and some "virtual" devices. The latter sometimes have a ridiculously high number of (virtual) inputs and outputs.

On Mac OS X, you might get something similar to this:

```
    >>> sd.query_devices()
      0 Built-in Line Input, Core Audio (2 in, 0 out)
    > 1 Built-in Digital Input, Core Audio (2 in, 0 out)
    < 2 Built-in Output, Core Audio (0 in, 2 out)
      3 Built-in Line Output, Core Audio (0 in, 2 out)
      4 Built-in Digital Output, Core Audio (0 in, 2 out)
```

sounddevice.**query_hostapis**(*index=None*)

Return information about available host APIs.

Parameters **index** (*int, optional*) – If specified, information about only the given host API *index* is returned in a single dictionary.

Returns

dict or tuple of dict – A dictionary with information about the given host API *index* or – if no *index* was specified – a tuple containing one dictionary for each available host API. The dictionaries have the following keys:

'name' The name of the host API.

'devices' A list of device IDs belonging to the host API. Use *query_devices()* to get information about a device.

'default_input_device','default_output_device' The device ID of the default input/output device of the host API. If no default input/output device exists for the given host API, this is -1.

Note: The overall default device(s) – which can be overwritten by assigning to *default.device* – take(s) precedence over *default.hostapi* and the information in the abovementioned dictionaries.

See also:

query_devices()

sounddevice.**check_input_settings**(*device=None*, *channels=None*, *dtype=None*, *samplerate=None*)

Check if given input device settings are supported.

All parameters are optional, *default* settings are used for any unspecified parameters. If the settings are supported, the function does nothing; if not, an exception is raised.

Parameters

- **device** (*int or str, optional*) – Device ID or device name substring, see *default.device*.
- **channels** (*int, optional*) – Number of input channels, see *default.channels*.
- **dtype** (*str or numpy.dtype, optional*) – Data type for input samples, see *default.dtype*.

- **samplerate** (*float, optional*) – Sampling frequency, see *default.samplerate*.

sounddevice.**check_output_settings**(*device=None*, *channels=None*, *dtype=None*, *samplerate=None*)

Check if given output device settings are supported.

Same as *check_input_settings()*, just for output device settings.

sounddevice.**sleep**(*msec*)

Put the caller to sleep for at least *msec* milliseconds.

The function may sleep longer than requested so don't rely on this for accurate musical timing.

sounddevice.**get_portaudio_version**()

Get version information for the PortAudio library.

Returns the release number and a textual description of the current PortAudio build, e.g.

```
(1899, 'PortAudio V19-devel (built Feb 15 2014 23:28:00)')
```

class sounddevice.**default**

Get/set defaults for the *sounddevice* module.

The attributes *device*, *channels*, *dtype* and *latency* accept single values which specify the given property for both input and output. However, if the property differs between input and output, pairs of values can be used, where the first value specifies the input and the second value specifies the output. All other attributes are always single values.

Examples

```
>>> import sounddevice as sd
>>> sd.default.samplerate = 48000
>>> sd.default.dtype
['float32', 'float32']
```

Different values for input and output:

```
>>> sd.default.channels = 1, 2
```

A single value sets both input and output at the same time:

```
>>> sd.default.device = 5
>>> sd.default.device
[5, 5]
```

An attribute can be set to the "factory default" by assigning `None`:

```
>>> sd.default.samplerate = None
>>> sd.default.device = None, 4
```

Use *reset()* to reset all attributes:

```
>>> sd.default.reset()
```

device = (None, None)

Index or query string of default input/output device.

If not overwritten, this is queried from PortAudio.

If a string is given, the device is selected which contains all space-separated parts in the right order. Each device string contains the name of the corresponding host API in the end. The string comparison is case-insensitive.

See also:

query_devices()

channels = (None, None)
Number of input/output channels.

The maximum number of channels for a given device can be found out with *query_devices()*.

dtype = ('float32', 'float32')
Data type used for input/output samples.

The types 'float32', 'int32', 'int16', 'int8' and 'uint8' can be used for all streams and functions. Additionally, *play()*, *rec()* and *playrec()* support 'float64' (for convenience, data is merely converted from/to 'float32') and *RawInputStream*, *RawOutputStream* and *RawStream* support 'int24' (packed 24 bit format – *not* supported in NumPy!).

If NumPy is available, the corresponding numpy.dtype[16] objects can be used as well.

The floating point representations 'float32' and 'float64' use +1.0 and -1.0 as the maximum and minimum values, respectively. 'uint8' is an unsigned 8 bit format where 128 is considered "ground".

latency = ('high', 'high')
Suggested input/output latency in seconds.

The special values 'low' and 'high' can be used to select the default low/high latency of the chosen device. 'high' is typically more robust (i.e. buffer under-/overflows are less likely), but the latency may be too large for interactive applications.

See also:

query_devices()

samplerate = None
Sampling frequency in Hertz (= frames per second).

See also:

query_devices()

blocksize = 0
See the *blocksize* argument of *Stream*.

clip_off = False
Disable clipping.

Set to True to disable default clipping of out of range samples.

dither_off = False
Disable dithering.

Set to True to disable default dithering.

never_drop_input = False
Set behavior for input overflow of full-duplex streams.

Set to True to request that where possible a full duplex stream will not discard overflowed input samples without calling the stream callback. This flag is only valid for full-duplex callback streams (i.e. only *Stream* and *RawStream* and only if *callback* was specified; this includes *playrec()*) and only when used in combination with *blocksize=0* (the default). Using this flag incorrectly results in an error being raised.

prime_output_buffers_using_stream_callback = False
How to fill initial output buffers.

Set to True to call the stream callback to fill initial output buffers, rather than the default behavior of priming the buffers with zeros (silence). This flag has no effect for input-only (*InputStream* and *RawInputStream*) and blocking read/write streams (i.e. if *callback* wasn't specified).

[16] http://docs.scipy.org/doc/numpy/reference/generated/numpy.dtype.html#numpy.dtype

hostapi
Index of the default host API (read-only).

reset()
Reset all attributes to their "factory default".

class sounddevice.**Stream**(*samplerate=None*, *blocksize=None*, *device=None*, *channels=None*, *dtype=None*, *latency=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)
Open a stream for input and output.

To open an input-only or output-only stream use *InputStream* or *OutputStream*, respectively. If you want to handle audio data as buffer objects instead of NumPy arrays, use *RawStream*, *RawInputStream* or *RawOutputStream*.

A single stream can provide multiple channels of real-time streaming audio input and output to a client application. A stream provides access to audio hardware represented by one or more devices. Depending on the underlying Host API, it may be possible to open multiple streams using the same device, however this behavior is implementation defined. Portable applications should assume that a device may be simultaneously used by at most one stream.

The arguments *device*, *channels*, *dtype* and *latency* can be either single values (which will be used for both input and output parameters) or pairs of values (where the first one is the value for the input and the second one for the output).

All arguments are optional, the values for unspecified parameters are taken from the *default* object. If one of the values of a parameter pair is None, the corresponding value from *default* will be used instead.

The created stream is inactive (see *active*, *stopped*). It can be started with *start()*.

Every stream object is also a context manager[17], i.e. it can be used in a with statement[18] to automatically call *start()* in the beginning of the statement and *stop()* and *close()* on exit.

Parameters

- **samplerate** (*float, optional*) – The desired sampling frequency (for both input and output). The default value can be changed with *default.samplerate*.
- **blocksize** (*int, optional*) – The number of frames passed to the stream callback function, or the preferred block granularity for a blocking read/write stream. The special value *blocksize=0* (which is the default) may be used to request that the stream callback will receive an optimal (and possibly varying) number of frames based on host requirements and the requested latency settings. The default value can be changed with *default.blocksize*.

  Note: With some host APIs, the use of non-zero *blocksize* for a callback stream may introduce an additional layer of buffering which could introduce additional latency. PortAudio guarantees that the additional latency will be kept to the theoretical minimum however, it is strongly recommended that a non-zero *blocksize* value only be used when your algorithm requires a fixed number of frames per stream callback.

- **device** (*int or str or pair thereof, optional*) – Device index(es) or query string(s) specifying the device(s) to be used. The default value(s) can be changed with *default.device*.
- **channels** (*int or pair of int, optional*) – The number of channels of sound to be delivered to the stream callback or accessed by *read()* or *write()*. It can range from 1 to the value of 'max_input_channels'/'max_output_channels' in the dict returned by *query_devices()*. By default, the maximum possible number of channels for the selected device is used (which may not be what

[17] http://docs.python.org/3/reference/datamodel.html#context-managers
[18] http://docs.python.org/3/reference/compound_stmts.html#with

you want; see *query_devices()*). The default value(s) can be changed with *default.channels*.

- **dtype** (*str or numpy.dtype or pair thereof, optional*) – The sample format of the numpy.ndarray[19] provided to the stream callback, *read()* or *write()*. It may be any of *float32*, *int32*, *int16*, *int8*, *uint8*. See numpy.dtype[20]. The *float64* data type is not supported, this is only supported for convenience in *play()*/*rec()*/*playrec()*. The packed 24 bit format `'int24'` is only supported in the "raw" stream classes, see *RawStream*. The default value(s) can be changed with *default.dtype*.

- **latency** (*float or {'low', 'high'} or pair thereof, optional*) – The desired latency in seconds. The special values `'low'` and `'high'` (latter being the default) select the default low and high latency, respectively (see *query_devices()*). The default value(s) can be changed with *default.latency*. Where practical, implementations should configure their latency based on this parameter, otherwise they may choose the closest viable latency instead. Unless the suggested latency is greater than the absolute upper limit for the device, implementations should round the *latency* up to the next practical value – i.e. to provide an equal or higher latency wherever possible. Actual latency values for an open stream may be retrieved using the *latency* attribute.

- **callback** (*callable, optional*) – User-supplied function to consume, process or generate audio data in response to requests from an *active* stream. When a stream is running, PortAudio calls the stream callback periodically. The callback function is responsible for processing and filling input and output buffers, respectively.

  If no *callback* is given, the stream will be opened in "blocking read/write" mode. In blocking mode, the client can receive sample data using *read()* and write sample data using *write()*, the number of frames that may be read or written without blocking is returned by *read_available* and *write_available*, respectively.

  The callback must have this signature:

```
callback(indata: ndarray, outdata: ndarray, frames: int,
         time: CData, status: CallbackFlags) -> None
```

  The first and second argument are the input and output buffer, respectively, as two-dimensional numpy.ndarray[21] with one column per channel (i.e. with a shape of *(frames, channels)*) and with a data type specified by *dtype*. The output buffer contains uninitialized data and the *callback* is supposed to fill it with proper audio data. If no data is available, the buffer should be filled with zeros (e.g. by using `outdata.fill(0)`).

  Note: In Python, assigning to an identifier merely re-binds the identifier to another object, so this *will not work* as expected:

```
outdata = my_data  # Don't do this!
```

  To actually assign data to the buffer itself, you can use indexing, e.g.:

```
outdata[:] = my_data
```

  ... which fills the whole buffer, or:

```
outdata[:, 1] = my_channel_data
```

  ... which only fills one channel.

  ---

[19] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[20] http://docs.scipy.org/doc/numpy/reference/generated/numpy.dtype.html#numpy.dtype
[21] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

The third argument holds the number of frames to be processed by the stream callback. This is the same as the length of the input and output buffers.

The forth argument provides a CFFI structure with timestamps indicating the ADC capture time of the first sample in the input buffer (*time.inputBufferAdcTime*), the DAC output time of the first sample in the output buffer (*time.outputBufferDacTime*) and the time the callback was invoked (*time.currentTime*). These time values are expressed in seconds and are synchronised with the time base used by `time` for the associated stream.

The fifth argument is a `CallbackFlags` instance indicating whether input and/or output buffers have been inserted or will be dropped to overcome underflow or overflow conditions.

If an exception is raised in the *callback*, it will not be called again. If `CallbackAbort` is raised, the stream will finish as soon as possible. If `CallbackStop` is raised, the stream will continue until all buffers generated by the callback have been played. This may be useful in applications such as soundfile players where a specific duration of output is required. If another exception is raised, its traceback is printed to `sys.stderr`[22]. Exceptions are *not* propagated to the main thread, i.e. the main Python program keeps running as if nothing had happened.

Note: The *callback* must always fill the entire output buffer, no matter if or which exceptions are raised.

If no exception is raised in the *callback*, it automatically continues to be called until `stop()`, `abort()` or `close()` are used to stop the stream.

The PortAudio stream callback runs at very high or real-time priority. It is required to consistently meet its time deadlines. Do not allocate memory, access the file system, call library functions or call other functions from the stream callback that may block or take an unpredictable amount of time to complete. With the exception of `cpu_load` it is not permissible to call PortAudio API functions from within the stream callback.

In order for a stream to maintain glitch-free operation the callback must consume and return audio data faster than it is recorded and/or played. PortAudio anticipates that each callback invocation may execute for a duration approaching the duration of *frames* audio frames at the stream's sampling frequency. It is reasonable to expect to be able to utilise 70% or more of the available CPU time in the PortAudio callback. However, due to buffer size adaption and other factors, not all host APIs are able to guarantee audio stability under heavy CPU load with arbitrary fixed callback buffer sizes. When high callback CPU utilisation is required the most robust behavior can be achieved by using *blocksize=0*.

- **finished_callback** (*callable, optional*) – User-supplied function which will be called when the stream becomes inactive (i.e. once a call to `stop()` will not block).

  A stream will become inactive after the stream callback raises an exception or when `stop()` or `abort()` is called. For a stream providing audio output, if the stream callback raises `CallbackStop`, or `stop()` is called, the stream finished callback will not be called until all generated sample data has been played. The callback must have this signature:

```
finished_callback() -> None
```

- **clip_off** (*bool, optional*) – See `default.clip_off`.
- **dither_off** (*bool, optional*) – See `default.dither_off`.
- **never_drop_input** (*bool, optional*) – See `default.never_drop_input`.

[22] http://docs.python.org/3/library/sys.html#sys.stderr

- **prime_output_buffers_using_stream_callback** (*bool, optional*) – See *default.prime_output_buffers_using_stream_callback*.

abort()
Terminate audio processing immediately.

This does not wait for pending buffers to complete.

See also:

start(), *stop()*

active
`True` when the stream is active, `False` otherwise.

A stream is active after a successful call to *start()*, until it becomes inactive either as a result of a call to *stop()* or *abort()*, or as a result of an exception raised in the stream callback. In the latter case, the stream is considered inactive after the last buffer has finished playing.

See also:

stopped

blocksize
Number of frames per block.

The special value 0 means that the blocksize can change between blocks. See the *blocksize* argument of *Stream*.

channels
The number of input/output channels.

close(*ignore_errors=True*)
Close the stream.

If the audio stream is active any pending buffers are discarded as if *abort()* had been called.

cpu_load
CPU usage information for the stream.

The "CPU Load" is a fraction of total CPU time consumed by a callback stream's audio processing routines including, but not limited to the client supplied stream callback. This function does not work with blocking read/write streams.

This may be used in the stream callback function or in the application. It provides a floating point value, typically between 0.0 and 1.0, where 1.0 indicates that the stream callback is consuming the maximum number of CPU cycles possible to maintain real-time operation. A value of 0.5 would imply that PortAudio and the stream callback was consuming roughly 50% of the available CPU time. The value may exceed 1.0. A value of 0.0 will always be returned for a blocking read/write stream, or if an error occurs.

device
IDs of the input/output device.

dtype
Data type of the audio samples.

See also:

default.dtype, *samplesize*

latency
The input/output latency of the stream in seconds.

This value provides the most accurate estimate of input/output latency available to the implementation. It may differ significantly from the *latency* value(s) passed to *Stream()*.

read(*frames*)
Read samples from the stream into a NumPy array.

The function doesn't return until all requested *frames* have been read – this may involve waiting for the operating system to supply the data (except if no more than *read_available* frames were requested).

This is the same as *RawStream.read()*, except that it returns a NumPy array instead of a plain Python buffer object.

Parameters **frames** (*int*) – The number of frames to be read. This parameter is not constrained to a specific range, however high performance applications will want to match this parameter to the *blocksize* parameter used when opening the stream.

Returns

- **data** (*numpy.ndarray*) – A two-dimensional numpy.ndarray[23] with one column per channel (i.e. with a shape of *(frames, channels)*) and with a data type specified by *dtype*.
- **overflowed** (*bool*) – True if input data was discarded by PortAudio after the previous call and before this call.

read_available

The number of frames that can be read without waiting.

Returns a value representing the maximum number of frames that can be read from the stream without blocking or busy waiting.

samplerate

The sampling frequency in Hertz (= frames per second).

In cases where the hardware sampling frequency is inaccurate and PortAudio is aware of it, the value of this field may be different from the *samplerate* parameter passed to *Stream*. If information about the actual hardware sampling frequency is not available, this field will have the same value as the *samplerate* parameter passed to *Stream*.

samplesize

The size in bytes of a single sample.

See also:

dtype

start()

Commence audio processing.

See also:

stop(), *abort()*

stop()

Terminate audio processing.

This waits until all pending audio buffers have been played before it returns.

See also:

start(), *abort()*

stopped

True when the stream is stopped, False otherwise.

A stream is considered to be stopped prior to a successful call to *start()* and after a successful call to *stop()* or *abort()*. If a stream callback is cancelled (by raising an exception) the stream is *not* considered to be stopped.

See also:

active

[23] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

time
The current stream time in seconds.

This is according to the same clock used to generate the timestamps passed with the *time* argument to the stream callback (see the *callback* argument of `Stream`). The time values are monotonically increasing and have unspecified origin.

This provides valid time values for the entire life of the stream, from when the stream is opened until it is closed. Starting and stopping the stream does not affect the passage of time as provided here.

This time may be used for synchronizing other events to the audio stream, for example synchronizing audio to MIDI.

write(*data*)
Write samples to the stream.

This function doesn't return until the entire buffer has been consumed – this may involve waiting for the operating system to consume the data (except if *data* contains no more than `write_available` frames).

This is the same as `RawStream.write()`, except that it expects a NumPy array instead of a plain Python buffer object.

Parameters **data** (*array_like*) – A two-dimensional array-like object with one column per channel (i.e. with a shape of *(frames, channels)*) and with a data type specified by `dtype`. A one-dimensional array can be used for mono data. The array layout must be C-contiguous (see `numpy.ascontiguousarray()`[24]).

The length of the buffer is not constrained to a specific range, however high performance applications will want to match this parameter to the *blocksize* parameter used when opening the stream.

Returns **underflowed** (*bool*) – `True` if additional output data was inserted after the previous call and before this call.

write_available
The number of frames that can be written without waiting.

Returns a value representing the maximum number of frames that can be written to the stream without blocking or busy waiting.

class sounddevice.**InputStream**(*samplerate=None*, *blocksize=None*, *device=None*, *channels=None*, *dtype=None*, *latency=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)
Open an input stream.

This has the same methods and attributes as `Stream`, except `write()` and `write_available`. Furthermore, the stream callback is expected to have a different signature (see below).

Parameters **callback** (*callable*) – User-supplied function to consume audio in response to requests from an active stream. The callback must have this signature:

```
callback(indata: numpy.ndarray, frames: int,
         time: CData, status: CallbackFlags) -> None
```

The arguments are the same as in the *callback* parameter of `Stream`, except that *outdata* is missing.

See also:

`Stream`, `RawInputStream`

[24] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ascontiguousarray.html#numpy.ascontiguousarray

class sounddevice.**OutputStream**(*samplerate=None*, *blocksize=None*, *device=None*, *channels=None*, *dtype=None*, *latency=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)

Open an output stream.

This has the same methods and attributes as *Stream*, except *read()* and *read_available*. Furthermore, the stream callback is expected to have a different signature (see below).

Parameters **callback** (*callable*) – User-supplied function to generate audio data in response to requests from an active stream. The callback must have this signature:

```
callback(outdata: numpy.ndarray, frames: int,
         time: CData, status: CallbackFlags) -> None
```

The arguments are the same as in the *callback* parameter of *Stream*, except that *indata* is missing.

See also:

Stream, *RawOutputStream*

class sounddevice.**RawStream**(*samplerate=None*, *blocksize=None*, *device=None*, *channels=None*, *dtype=None*, *latency=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)

Open a "raw" input/output stream.

This is the same as *Stream*, except that the *callback* function and *read()*/*write()* work on plain Python buffer objects instead of on NumPy arrays. NumPy is not necessary to use this.

To open "raw" input-only or output-only stream use *RawInputStream* or *RawOutputStream*, respectively. If you want to handle audio data as NumPy arrays instead of buffer objects, use *Stream*, *InputStream* or *OutputStream*.

Parameters

- **dtype** (*str or pair of str*) – The sample format of the buffers provided to the stream callback, *read()* or *write()*. In addition to the formats supported by *Stream* (`'float32'`, `'int32'`, `'int16'`, `'int8'`, `'uint8'`), this also supports `'int24'`, i.e. packed 24 bit format. The default value can be changed with *default.dtype*. See also *Stream.samplesize*.
- **callback** (*callable*) – User-supplied function to consume, process or generate audio data in response to requests from an active stream. The callback must have this signature:

```
callback(indata: buffer, outdata: buffer, frames: int,
         time: CData, status: CallbackFlags) -> None
```

The arguments are the same as in the *callback* parameter of *Stream*, except that *indata* and *outdata* are plain Python buffer objects instead of NumPy arrays.

See also:

RawInputStream, *RawOutputStream*, *Stream*

read(*frames*)

Read samples from the stream into a buffer.

This is the same as *Stream.read()*, except that it returns a plain Python buffer object instead of a NumPy array. NumPy is not necessary to use this.

Parameters **frames** (*int*) – The number of frames to be read. See *Stream.read()*.

Returns

- **data** (*buffer*) – A buffer of interleaved samples. The buffer contains samples in the format specified by the *dtype* parameter used to open the stream, and the number of channels specified by *channels*. See also `Stream.samplesize`.
- **overflowed** (*bool*) – See `Stream.read()`.

write(*data*)

Write samples to the stream.

This is the same as `Stream.write()`, except that it expects a plain Python buffer object instead of a NumPy array. NumPy is not necessary to use this.

Parameters **data** (*buffer or bytes or iterable of int*) – A buffer of interleaved samples. The buffer contains samples in the format specified by the *dtype* argument used to open the stream, and the number of channels specified by *channels*. The length of the buffer is not constrained to a specific range, however high performance applications will want to match this parameter to the *blocksize* parameter used when opening the stream. See also `Stream.samplesize`.

Returns **underflowed** (*bool*) – See `Stream.write()`.

class sounddevice.**RawInputStream**(*samplerate=None*, *blocksize=None*, *device=None*, *channels=None*, *dtype=None*, *latency=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)

Open a "raw" input stream.

This is the same as `InputStream`, except that the *callback* function and `read()` work on plain Python buffer objects instead of on NumPy arrays. NumPy is not necessary to use this.

Parameters

- **dtype** (*str*) – See `RawStream`.
- **callback** (*callable*) – User-supplied function to consume audio data in response to requests from an active stream. The callback must have this signature:

```
callback(indata: buffer, frames: int,
         time: CData, status: CallbackFlags) -> None
```

The arguments are the same as in the *callback* parameter of `RawStream`, except that *outdata* is missing.

See also:

`RawStream`, `Stream`

class sounddevice.**RawOutputStream**(*samplerate=None*, *blocksize=None*, *device=None*, *channels=None*, *dtype=None*, *latency=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)

Open a "raw" output stream.

This is the same as `OutputStream`, except that the *callback* function and `write()` work on plain Python buffer objects instead of on NumPy arrays. NumPy is not necessary to use this.

Parameters

- **dtype** (*str*) – See `RawStream`.
- **callback** (*callable*) – User-supplied function to generate audio data in response to requests from an active stream. The callback must have this signature:

```
callback(outdata: buffer, frames: int,
         time: CData, status: CallbackFlags) -> None
```

The arguments are the same as in the *callback* parameter of *RawStream*, except that *indata* is missing.

See also:

RawStream, *Stream*

class sounddevice.**DeviceList**
A list with information about all available audio devices.

This class is not meant to be instantiated by the user. Instead, it is returned by *query_devices()*. It contains a dictionary for each available device, holding the keys described in *query_devices()*.

This class has a special string representation that is shown as return value of *query_devices()* if used in an interactive Python session. It will also be shown when using the `print()`[25] function. Furthermore, it can be obtained with `repr()`[26] and `str()`[27].

class sounddevice.**CallbackFlags** (*flags=0*)
Flag bits for the *status* argument to a stream *callback*.

See also:

Stream

Examples

This can be used to collect the errors of multiple *status* objects:

```
>>> import sounddevice as sd
>>> errors = sd.CallbackFlags()
>>> errors |= status1
>>> errors |= status2
>>> errors |= status3
>>> # and so on ...
>>> errors.input_overflow
True
```

input_underflow
Input underflow.

In a stream opened with *blocksize=0*, indicates that input data is all silence (zeros) because no real data is available. In a stream opened with a non-zero *blocksize*, it indicates that one or more zero samples have been inserted into the input buffer to compensate for an input underflow.

input_overflow
Input overflow.

In a stream opened with *blocksize=0*, indicates that data prior to the first sample of the input buffer was discarded due to an overflow, possibly because the stream callback is using too much CPU time. Otherwise indicates that data prior to one or more samples in the input buffer was discarded.

output_underflow
Output underflow.

Indicates that output data (or a gap) was inserted, possibly because the stream callback is using too much CPU time.

output_overflow
Output overflow.

Indicates that output data will be discarded because no room is available.

[25] http://docs.python.org/3/library/functions.html#print
[26] http://docs.python.org/3/library/functions.html#repr
[27] http://docs.python.org/3/library/stdtypes.html#str

priming_output
Priming output.

Some of all of the output data will be used to prime the stream, input data may be zero.

class sounddevice.**CallbackStop**
Exception to be raised by the user to stop callback processing.

If this is raised in the stream callback, the callback will not be invoked anymore (but all pending audio buffers will be played).

See also:

CallbackAbort, *Stream.stop()*, *Stream*

class sounddevice.**CallbackAbort**
Exception to be raised by the user to abort callback processing.

If this is raised in the stream callback, all pending buffers are discarded and the callback will not be invoked anymore.

See also:

CallbackStop, *Stream.abort()*, *Stream*

class sounddevice.**PortAudioError**
This exception will be raised on PortAudio errors.

6 Version History

Version 0.3.2 (2016-03-16):

- `mapping=[1]` works now on all host APIs
- Example application `plot_input.py` showing the live microphone signal(s)
- Device substrings are now allowed in *sounddevice.query_devices()*

Version 0.3.1 (2016-01-04):

- Add *sounddevice.check_input_settings()* and *sounddevice.check_output_settings()*
- Send PortAudio output to `/dev/null` (on Linux and OSX)

Version 0.3.0 (2015-10-28):

- Remove `sounddevice.print_devices()`, *sounddevice.query_devices()* can be used instead, since it now returns a *sounddevice.DeviceList* object.

Version 0.2.2 (2015-10-21):

- Devices can now be selected by substrings of device name and host API name

Version 0.2.1 (2015-10-08):

- Example applications `wire.py` (based on PortAudio's `patest_wire.c`) and `spectrogram.py` (based on code by Mauris Van Hauwe)

Version 0.2.0 (2015-07-03):

- Support for wheels including a dylib for Mac OS X and DLLs for Windows. The code for creating the wheels is largely taken from PySoundFile[28].
- Remove logging (this seemed too intrusive)
- Return callback status from *sounddevice.wait()* and add the new function *sounddevice.get_status()*

[28] https://github.com/bastibe/PySoundFile/

- `sounddevice.playrec()`: Rename the arguments *input_channels* and *input_dtype* to *channels* and *dtype*, respectively

Version 0.1.0 (2015-06-20): Initial release. Some ideas are taken from PySoundCard[29]. Thanks to Bastian Bechtold for many fruitful discussions during the development of several features which *python-sounddevice* inherited from there.

[29] https://github.com/bastibe/PySoundCard/