python-sounddevice

Release 0.5.2

Matthias Geier

2025-05-16

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.

The `sounddevice` module is available for Linux, macOS and Windows.

Documentation:
https://python-sounddevice.readthedocs.io/

Source code repository and issue tracker:
https://github.com/spatialaudio/python-sounddevice/

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

1 Installation

First of all, you'll need Python[4]. There are many distributions of Python available, anyone where CFFI[5] is supported should work.

You can get the latest `sounddevice` release from PyPI[6] (using `pip`). But first, it is recommended to create a virtual environment (e.g. using python3 -m venv[7] or conda create[8]). After activating the environment, the `sounddevice` module can be installed with:

```
python -m pip install sounddevice
```

If you have already installed an old version of the module in the environment, you can use the `--upgrade` flag to get the newest release. To un-install, use:

```
python -m pip uninstall sounddevice
```

If you want to try the very latest development version of the `sounddevice` module, have a look at the section about *Contributing*.

If you install the `sounddevice` module with `pip` on macOS or Windows, the PortAudio[9] library will be installed automagically. On other platforms, you might have to install PortAudio with your package manager (the package might be called `libportaudio2` or similar).

Note

If you install PortAudio with a package manager (including `conda`), it will likely override the version installed with `pip`.

The NumPy[10] library 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 needed – and not installed already – NumPy can be installed like this:

[1] https://www.python.org/
[2] http://www.portaudio.com/
[3] https://numpy.org/
[4] https://www.python.org/
[5] https://cffi.readthedocs.io/
[6] https://pypi.org/project/sounddevice/
[7] https://docs.python.org/3/library/venv.html
[8] https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html#creating-environments
[9] http://www.portaudio.com/
[10] https://numpy.org/

```
python -m pip install numpy
```

1.1 ASIO Support

Installing the `sounddevice` module with `pip` (on Windows) will provide two PortAudio[11] DLLs, with and without ASIO support. By default, the DLL *without* ASIO support is loaded (because of the problems mentioned in issue #496[12]). To load the DLL *with* ASIO support, the environment variable `SD_ENABLE_ASIO` has to be set *before* importing the `sounddevice` module, for example like this:

```
import os

# Set environment variable before importing sounddevice. Value is not important.
os.environ["SD_ENABLE_ASIO"] = "1"

import sounddevice as sd

print(sd.query_hostapis())
```

Note

This will only work if the `sounddevice` module has been installed via `pip`. This will not work with the `conda` package (see below).

Note

This will not work if a custom `portaudio.dll` is present in the `%PATH%` (as described in the following section).

1.2 Custom PortAudio Library

If you want to use a different version of the PortAudio library (maybe a development version or a version with different features selected), you can rename the library to `libportaudio.so` (Linux) or `libportaudio.dylib` (macOS) and move it to `/usr/local/lib`. On Linux, you might have to run `sudo ldconfig` after that, for the library to be found. On Windows, you can rename the library to `portaudio.dll` and move it to any directory in your `%PATH%`. In case of doubt you should create a fresh directory for your library and add that to your `PATH` variable.

1.3 Alternative Packages

If you are using the `conda` package manager (e.g. with miniforge[13]), you can install the `sounddevice` module from the `conda-forge` channel:

```
conda install -c conda-forge python-sounddevice
```

You can of course also use `mamba` if `conda` is too slow.

Note

The PortAudio package on `conda-forge` doesn't have ASIO support, see https://github.com/conda-forge/portaudio-feedstock/issues/9.

[11] http://www.portaudio.com/
[12] https://github.com/spatialaudio/python-sounddevice/issues/496
[13] https://github.com/conda-forge/miniforge

There are also packages for several other package managers:

https://repology.org/metapackage/python:sounddevice

2 Usage

First, import the module:

```
import sounddevice as sd
```

2.1 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 *play()*:

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

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

```
sd.stop()
```

If you want to block the Python interpreter until playback is finished, you can use *wait()*:

```
sd.wait()
```

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

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

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

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

> **Note**
>
> If you don't specify the correct sampling frequency, the sound might be played back too slow or too fast!

2.2 Recording

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

```
duration = 10.5  # seconds
myrecording = sd.rec(int(duration * fs), samplerate=fs, channels=2)
```

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

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

After that, you can drop the additional arguments:

```
myrecording = sd.rec(int(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 *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.

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

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

2.3 Simultaneous Playback and Recording

To play back an array and record at the same time, you can use *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 *default.channels*.

2.4 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 *query_devices()* to get a list of supported devices. The same list can be obtained from a terminal by typing the command

```
python3 -m sounddevice
```

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

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

2.5 Callback Streams

The aforementioned convenience functions *play()*, *rec()* and *playrec()* (as well as the related functions *wait()*, *stop()*, *get_status()* and *get_stream()*) are designed for small scripts and interactive use (e.g. in a Jupyter[14] notebook). They are supposed to be simple and convenient, but their use cases are quite limited.

If you need more control (e.g. continuous recording, realtime processing, ...), you should use the lower-level "stream" classes (e.g. *Stream*, *InputStream*, *RawInputStream*), either with the "non-blocking" callback interface or with the "blocking" *Stream.read()* and *Stream.write()* methods, see *Blocking Read/Write Streams*.

As an example for the "non-blocking" interface, the following code creates a *Stream* with a callback function that obtains audio data from the input channels and simply forwards everything to the output channels (be careful with the output volume, because this might cause acoustic feedback if your microphone is close to your loudspeakers):

[14] https://jupyter.org/

```
import sounddevice as sd
duration = 5.5  # seconds

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

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

The same thing can be done with *RawStream* (NumPy[15] doesn't have to be installed):

```
import sounddevice as sd
duration = 5.5  # seconds

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

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

Note

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

You can of course extend the callback functions to do arbitrarily more complicated stuff. You can also use streams without inputs (e.g. *OutputStream*) or streams without outputs (e.g. *InputStream*).

See *Example Programs* for more examples.

2.6 Blocking Read/Write Streams

Instead of using a callback function, you can also use the "blocking" methods *Stream.read()* and *Stream.write()* (and of course the corresponding methods in *InputStream*, *OutputStream*, *RawStream*, *RawInputStream* and *RawOutputStream*).

3 Example Programs

Most of these examples use the `argparse`[16] module to handle command line arguments. To show a help text explaining all available arguments, use the `--help` argument.

For example:

```
python play_file.py --help
```

3.1 Play a Sound File

`play_file.py`[17]

[15] https://numpy.org/
[16] https://docs.python.org/3/library/argparse.html#module-argparse
[17] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/play_file.py

```
#!/usr/bin/env python3
"""Load an audio file into memory and play its contents.

NumPy and the soundfile module (https://python-soundfile.readthedocs.io/)
must be installed for this to work.

This example program loads the whole file into memory before starting
playback.
To play very long files, you should use play_long_file.py instead.

This example could simply be implemented like this::

    import sounddevice as sd
    import soundfile as sf

    data, fs = sf.read('my-file.wav')
    sd.play(data, fs)
    sd.wait()

... but in this example we show a more low-level implementation
using a callback stream.

"""
import argparse
import threading

import sounddevice as sd
import soundfile as sf


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    'filename', metavar='FILENAME',
    help='audio file to be played back')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='output device (numeric ID or substring)')
args = parser.parse_args(remaining)
```

(continues on next page)

(continued from previous page)

```
event = threading.Event()

try:
    data, fs = sf.read(args.filename, always_2d=True)

    current_frame = 0

    def callback(outdata, frames, time, status):
        global current_frame
        if status:
            print(status)
        chunksize = min(len(data) - current_frame, frames)
        outdata[:chunksize] = data[current_frame:current_frame + chunksize]
        if chunksize < frames:
            outdata[chunksize:] = 0
            raise sd.CallbackStop()
        current_frame += chunksize

    stream = sd.OutputStream(
        samplerate=fs, device=args.device, channels=data.shape[1],
        callback=callback, finished_callback=event.set)
    with stream:
        event.wait()  # Wait until playback is finished
except KeyboardInterrupt:
    parser.exit('\nInterrupted by user')
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.2 Play a Very Long Sound File

play_long_file.py[18]

```
#!/usr/bin/env python3
"""Play an audio file using a limited amount of memory.

The soundfile module (https://python-soundfile.readthedocs.io/) must be
installed for this to work.

In contrast to play_file.py, which loads the whole file into memory
before starting playback, this example program only holds a given number
of audio blocks in memory and is therefore able to play files that are
larger than the available RAM.

This example is implemented using NumPy, see play_long_file_raw.py
for a version that doesn't need NumPy.

"""
import argparse
import queue
import sys
import threading

import sounddevice as sd
```

(continues on next page)

[18] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/play_long_file.py

(continued from previous page)

```
import soundfile as sf


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    'filename', metavar='FILENAME',
    help='audio file to be played back')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='output device (numeric ID or substring)')
parser.add_argument(
    '-b', '--blocksize', type=int, default=2048,
    help='block size (default: %(default)s)')
parser.add_argument(
    '-q', '--buffersize', type=int, default=20,
    help='number of blocks used for buffering (default: %(default)s)')
args = parser.parse_args(remaining)
if args.blocksize == 0:
    parser.error('blocksize must not be zero')
if args.buffersize < 1:
    parser.error('buffersize must be at least 1')

q = queue.Queue(maxsize=args.buffersize)
event = threading.Event()


def callback(outdata, frames, time, status):
    assert frames == args.blocksize
    if status.output_underflow:
        print('Output underflow: increase blocksize?', file=sys.stderr)
        raise sd.CallbackAbort
    assert not status
    try:
        data = q.get_nowait()
    except queue.Empty as e:
        print('Buffer is empty: increase buffersize?', file=sys.stderr)
        raise sd.CallbackAbort from e
```

(continues on next page)

(continued from previous page)

```
    if len(data) < len(outdata):
        outdata[:len(data)] = data
        outdata[len(data):].fill(0)
        raise sd.CallbackStop
    else:
        outdata[:] = data


try:
    with sf.SoundFile(args.filename) as f:
        for _ in range(args.buffersize):
            data = f.read(args.blocksize)
            if not len(data):
                break
            q.put_nowait(data)  # Pre-fill queue
        stream = sd.OutputStream(
            samplerate=f.samplerate, blocksize=args.blocksize,
            device=args.device, channels=f.channels,
            callback=callback, finished_callback=event.set)
        with stream:
            timeout = args.blocksize * args.buffersize / f.samplerate
            while len(data):
                data = f.read(args.blocksize)
                q.put(data, timeout=timeout)
            event.wait()  # Wait until playback is finished
except KeyboardInterrupt:
    parser.exit('\nInterrupted by user')
except queue.Full:
    # A timeout occurred, i.e. there was an error in the callback
    parser.exit(1)
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.3 Play a Very Long Sound File without Using NumPy

play_long_file_raw.py[19]

```
#!/usr/bin/env python3
"""Play an audio file using a limited amount of memory.

This is the same as play_long_file.py, but implemented without using NumPy.

"""
import argparse
import queue
import sys
import threading

import sounddevice as sd
import soundfile as sf


def int_or_str(text):
    """Helper function for argument parsing."""
```

(continues on next page)

[19] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/play_long_file_raw.py

(continued from previous page)

```
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    'filename', metavar='FILENAME',
    help='audio file to be played back')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='output device (numeric ID or substring)')
parser.add_argument(
    '-b', '--blocksize', type=int, default=2048,
    help='block size (default: %(default)s)')
parser.add_argument(
    '-q', '--buffersize', type=int, default=20,
    help='number of blocks used for buffering (default: %(default)s)')
args = parser.parse_args(remaining)
if args.blocksize == 0:
    parser.error('blocksize must not be zero')
if args.buffersize < 1:
    parser.error('buffersize must be at least 1')

q = queue.Queue(maxsize=args.buffersize)
event = threading.Event()


def callback(outdata, frames, time, status):
    assert frames == args.blocksize
    if status.output_underflow:
        print('Output underflow: increase blocksize?', file=sys.stderr)
        raise sd.CallbackAbort
    assert not status
    try:
        data = q.get_nowait()
    except queue.Empty as e:
        print('Buffer is empty: increase buffersize?', file=sys.stderr)
        raise sd.CallbackAbort from e
    if len(data) < len(outdata):
        outdata[:len(data)] = data
        outdata[len(data):] = b'\x00' * (len(outdata) - len(data))
        raise sd.CallbackStop
    else:
```

(continues on next page)

(continued from previous page)

```
        outdata[:] = data


try:
    with sf.SoundFile(args.filename) as f:
        for _ in range(args.buffersize):
            data = f.buffer_read(args.blocksize, dtype='float32')
            if not data:
                break
            q.put_nowait(data)  # Pre-fill queue
        stream = sd.RawOutputStream(
            samplerate=f.samplerate, blocksize=args.blocksize,
            device=args.device, channels=f.channels, dtype='float32',
            callback=callback, finished_callback=event.set)
        with stream:
            timeout = args.blocksize * args.buffersize / f.samplerate
            while data:
                data = f.buffer_read(args.blocksize, dtype='float32')
                q.put(data, timeout=timeout)
            event.wait()  # Wait until playback is finished
except KeyboardInterrupt:
    parser.exit('\nInterrupted by user')
except queue.Full:
    # A timeout occurred, i.e. there was an error in the callback
    parser.exit(1)
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.4 Play a Web Stream

play_stream.py[20]

```
#!/usr/bin/env python3
"""Play a web stream.

ffmpeg-python (https://github.com/kkroening/ffmpeg-python) has to be installed.

If you don't know a stream URL, try http://icecast.spc.org:8000/longplayer
(see https://longplayer.org/ for a description).

"""
import argparse
import queue
import sys

import ffmpeg
import sounddevice as sd


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
```

(continues on next page)

[20] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/play_stream.py

(continued from previous page)

```
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    'url', metavar='URL',
    help='stream URL')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='output device (numeric ID or substring)')
parser.add_argument(
    '-b', '--blocksize', type=int, default=1024,
    help='block size (default: %(default)s)')
parser.add_argument(
    '-q', '--buffersize', type=int, default=20,
    help='number of blocks used for buffering (default: %(default)s)')
args = parser.parse_args(remaining)
if args.blocksize == 0:
    parser.error('blocksize must not be zero')
if args.buffersize < 1:
    parser.error('buffersize must be at least 1')

q = queue.Queue(maxsize=args.buffersize)

print('Getting stream information ...')

try:
    info = ffmpeg.probe(args.url)
except ffmpeg.Error as e:
    sys.stderr.buffer.write(e.stderr)
    parser.exit(e)

streams = info.get('streams', [])
if len(streams) != 1:
    parser.exit('There must be exactly one stream available')

stream = streams[0]

if stream.get('codec_type') != 'audio':
    parser.exit('The stream must be an audio stream')

channels = stream['channels']
samplerate = float(stream['sample_rate'])
```

(continues on next page)

(continued from previous page)

```
def callback(outdata, frames, time, status):
    assert frames == args.blocksize
    if status.output_underflow:
        print('Output underflow: increase blocksize?', file=sys.stderr)
        raise sd.CallbackAbort
    assert not status
    try:
        data = q.get_nowait()
    except queue.Empty as e:
        print('Buffer is empty: increase buffersize?', file=sys.stderr)
        raise sd.CallbackAbort from e
    assert len(data) == len(outdata)
    outdata[:] = data


try:
    print('Opening stream ...')
    process = ffmpeg.input(
        args.url
    ).output(
        'pipe:',
        format='f32le',
        acodec='pcm_f32le',
        ac=channels,
        ar=samplerate,
        loglevel='quiet',
    ).run_async(pipe_stdout=True)
    stream = sd.RawOutputStream(
        samplerate=samplerate, blocksize=args.blocksize,
        device=args.device, channels=channels, dtype='float32',
        callback=callback)
    read_size = args.blocksize * channels * stream.samplesize
    print('Buffering ...')
    for _ in range(args.buffersize):
        q.put_nowait(process.stdout.read(read_size))
    print('Starting Playback ...')
    with stream:
        timeout = args.blocksize * args.buffersize / samplerate
        while True:
            q.put(process.stdout.read(read_size), timeout=timeout)
except KeyboardInterrupt:
    parser.exit('\nInterrupted by user')
except queue.Full:
    # A timeout occurred, i.e. there was an error in the callback
    parser.exit(1)
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.5 Play a Sine Signal

play_sine.py[21]

```
#!/usr/bin/env python3
"""Play a sine signal."""
```

(continues on next page)

[21] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/play_sine.py

(continued from previous page)

```
import argparse
import sys

import numpy as np
import sounddevice as sd


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    'frequency', nargs='?', metavar='FREQUENCY', type=float, default=500,
    help='frequency in Hz (default: %(default)s)')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='output device (numeric ID or substring)')
parser.add_argument(
    '-a', '--amplitude', type=float, default=0.2,
    help='amplitude (default: %(default)s)')
args = parser.parse_args(remaining)

start_idx = 0

try:
    samplerate = sd.query_devices(args.device, 'output')['default_samplerate']

    def callback(outdata, frames, time, status):
        if status:
            print(status, file=sys.stderr)
        global start_idx
        t = (start_idx + np.arange(frames)) / samplerate
        t = t.reshape(-1, 1)
        outdata[:] = args.amplitude * np.sin(2 * np.pi * args.frequency * t)
        start_idx += frames

    with sd.OutputStream(device=args.device, channels=1, callback=callback,
                         samplerate=samplerate):
        print('#' * 80)
        print('press Return to quit')
```

(continues on next page)

(continued from previous page)

```
        print('#' * 80)
        input()
except KeyboardInterrupt:
    parser.exit('')
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.6 Input to Output Pass-Through

wire.py[22]

```
#!/usr/bin/env python3
"""Pass input directly to output.

https://github.com/PortAudio/portaudio/blob/master/test/patest_wire.c

"""
import argparse

import sounddevice as sd
import numpy  # Make sure NumPy is loaded before it is used in the callback
assert numpy  # avoid "imported but unused" message (W0611)


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    '-i', '--input-device', type=int_or_str,
    help='input device (numeric ID or substring)')
parser.add_argument(
    '-o', '--output-device', type=int_or_str,
    help='output device (numeric ID or substring)')
parser.add_argument(
    '-c', '--channels', type=int, default=2,
    help='number of channels')
parser.add_argument('--dtype', help='audio data type')
```

(continues on next page)

[22] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/wire.py

(continued from previous page)

```
parser.add_argument('--samplerate', type=float, help='sampling rate')
parser.add_argument('--blocksize', type=int, help='block size')
parser.add_argument('--latency', type=float, help='latency in seconds')
args = parser.parse_args(remaining)


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


try:
    with sd.Stream(device=(args.input_device, args.output_device),
                   samplerate=args.samplerate, blocksize=args.blocksize,
                   dtype=args.dtype, latency=args.latency,
                   channels=args.channels, callback=callback):
        print('#' * 80)
        print('press Return to quit')
        print('#' * 80)
        input()
except KeyboardInterrupt:
    parser.exit('')
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.7 Plot Microphone Signal(s) in Real-Time

plot_input.py[23]

```
#!/usr/bin/env python3
"""Plot the live microphone signal(s) with matplotlib.

Matplotlib and NumPy have to be installed.

"""
import argparse
import queue
import sys

from matplotlib.animation import FuncAnimation
import matplotlib.pyplot as plt
import numpy as np
import sounddevice as sd


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text
```

(continues on next page)

[23] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/plot_input.py

(continued from previous page)

```
parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    'channels', type=int, default=[1], nargs='*', metavar='CHANNEL',
    help='input channels to plot (default: the first)')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='input device (numeric ID or substring)')
parser.add_argument(
    '-w', '--window', type=float, default=200, metavar='DURATION',
    help='visible time slot (default: %(default)s ms)')
parser.add_argument(
    '-i', '--interval', type=float, default=30,
    help='minimum time between plot updates (default: %(default)s ms)')
parser.add_argument(
    '-b', '--blocksize', type=int, help='block size (in samples)')
parser.add_argument(
    '-r', '--samplerate', type=float, help='sampling rate of audio device')
parser.add_argument(
    '-n', '--downsample', type=int, default=10, metavar='N',
    help='display every Nth sample (default: %(default)s)')
args = parser.parse_args(remaining)
if any(c < 1 for c in args.channels):
    parser.error('argument CHANNEL: must be >= 1')
mapping = [c - 1 for c in args.channels]  # Channel numbers start with 1
q = queue.Queue()


def audio_callback(indata, frames, time, status):
    """This is called (from a separate thread) for each audio block."""
    if status:
        print(status, file=sys.stderr)
    # Fancy indexing with mapping creates a (necessary!) copy:
    q.put(indata[::args.downsample, mapping])


def update_plot(frame):
    """This is called by matplotlib for each plot update.

    Typically, audio callbacks happen more frequently than plot updates,
    therefore the queue tends to contain multiple blocks of audio data.

    """
    global plotdata
    while True:
        try:
```

(continues on next page)

(continued from previous page)

```
            data = q.get_nowait()
        except queue.Empty:
            break
        shift = len(data)
        plotdata = np.roll(plotdata, -shift, axis=0)
        plotdata[-shift:, :] = data
    for column, line in enumerate(lines):
        line.set_ydata(plotdata[:, column])
    return lines


try:
    if args.samplerate is None:
        device_info = sd.query_devices(args.device, 'input')
        args.samplerate = device_info['default_samplerate']

    length = int(args.window * args.samplerate / (1000 * args.downsample))
    plotdata = np.zeros((length, len(args.channels)))

    fig, ax = plt.subplots()
    lines = ax.plot(plotdata)
    if len(args.channels) > 1:
        ax.legend([f'channel {c}' for c in args.channels],
                  loc='lower left', ncol=len(args.channels))
    ax.axis((0, len(plotdata), -1, 1))
    ax.set_yticks([0])
    ax.yaxis.grid(True)
    ax.tick_params(bottom=False, top=False, labelbottom=False,
                   right=False, left=False, labelleft=False)
    fig.tight_layout(pad=0)

    stream = sd.InputStream(
        device=args.device, channels=max(args.channels),
        samplerate=args.samplerate, callback=audio_callback)
    ani = FuncAnimation(fig, update_plot, interval=args.interval, blit=True)
    with stream:
        plt.show()
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.8 Real-Time Text-Mode Spectrogram

spectrogram.py[24]

```
#!/usr/bin/env python3
"""Show a text-mode spectrogram using live microphone data."""
import argparse
import math
import shutil

import numpy as np
import sounddevice as sd

usage_line = ' press <enter> to quit, +<enter> or -<enter> to change scaling '
```

(continues on next page)

[24] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/spectrogram.py

(continued from previous page)

```
def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


try:
    columns, _ = shutil.get_terminal_size()
except AttributeError:
    columns = 80

parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__ + '\n\nSupported keys:' + usage_line,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    '-b', '--block-duration', type=float, metavar='DURATION', default=50,
    help='block size (default %(default)s milliseconds)')
parser.add_argument(
    '-c', '--columns', type=int, default=columns,
    help='width of spectrogram')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='input device (numeric ID or substring)')
parser.add_argument(
    '-g', '--gain', type=float, default=10,
    help='initial gain factor (default %(default)s)')
parser.add_argument(
    '-r', '--range', type=float, nargs=2,
    metavar=('LOW', 'HIGH'), default=[100, 2000],
    help='frequency range (default %(default)s Hz)')
args = parser.parse_args(remaining)
low, high = args.range
if high <= low:
    parser.error('HIGH must be greater than LOW')

# Create a nice output gradient using ANSI escape sequences.
# Stolen from https://gist.github.com/maurisvh/df919538bcef391bc89f
colors = 30, 34, 35, 91, 93, 97
chars = ' :%#\t#%:'
gradient = []
for bg, fg in zip(colors, colors[1:]):
    for char in chars:
        if char == '\t':
```

(continues on next page)

(continued from previous page)

```
            bg, fg = fg, bg
        else:
            gradient.append(f'\x1b[{fg};{bg + 10}m{char}')

try:
    samplerate = sd.query_devices(args.device, 'input')['default_samplerate']

    delta_f = (high - low) / (args.columns - 1)
    fftsize = math.ceil(samplerate / delta_f)
    low_bin = math.floor(low / delta_f)

    def callback(indata, frames, time, status):
        if status:
            text = ' ' + str(status) + ' '
            print('\x1b[34;40m', text.center(args.columns, '#'),
                  '\x1b[0m', sep='')
        if any(indata):
            magnitude = np.abs(np.fft.rfft(indata[:, 0], n=fftsize))
            magnitude *= args.gain / fftsize
            line = (gradient[int(np.clip(x, 0, 1) * (len(gradient) - 1))]
                    for x in magnitude[low_bin:low_bin + args.columns])
            print(*line, sep='', end='\x1b[0m\n')
        else:
            print('no input')

    with sd.InputStream(device=args.device, channels=1, callback=callback,
                        blocksize=int(samplerate * args.block_duration / 1000),
                        samplerate=samplerate):
        while True:
            response = input()
            if response in ('', 'q', 'Q'):
                break
            for ch in response:
                if ch == '+':
                    args.gain *= 2
                elif ch == '-':
                    args.gain /= 2
                else:
                    print('\x1b[31;40m', usage_line.center(args.columns, '#'),
                          '\x1b[0m', sep='')
                    break
except KeyboardInterrupt:
    parser.exit('Interrupted by user')
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.9 Recording with Arbitrary Duration

rec_unlimited.py[25]

```
#!/usr/bin/env python3
"""Create a recording with arbitrary duration.

The soundfile module (https://python-soundfile.readthedocs.io/)
```

(continues on next page)

[25] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/rec_unlimited.py

(continued from previous page)

```
has to be installed!

"""
import argparse
import tempfile
import queue
import sys

import sounddevice as sd
import soundfile as sf
import numpy  # Make sure NumPy is loaded before it is used in the callback
assert numpy  # avoid "imported but unused" message (W0611)


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    'filename', nargs='?', metavar='FILENAME',
    help='audio file to store recording to')
parser.add_argument(
    '-d', '--device', type=int_or_str,
    help='input device (numeric ID or substring)')
parser.add_argument(
    '-r', '--samplerate', type=int, help='sampling rate')
parser.add_argument(
    '-c', '--channels', type=int, default=1, help='number of input channels')
parser.add_argument(
    '-t', '--subtype', type=str, help='sound file subtype (e.g. "PCM_24")')
args = parser.parse_args(remaining)

q = queue.Queue()


def callback(indata, frames, time, status):
    """This is called (from a separate thread) for each audio block."""
    if status:
        print(status, file=sys.stderr)
    q.put(indata.copy())
```

(continues on next page)

(continued from previous page)

```
try:
    if args.samplerate is None:
        device_info = sd.query_devices(args.device, 'input')
        # soundfile expects an int, sounddevice provides a float:
        args.samplerate = int(device_info['default_samplerate'])
    if args.filename is None:
        args.filename = tempfile.mktemp(prefix='delme_rec_unlimited_',
                                        suffix='.wav', dir='')

    # Make sure the file is opened before recording anything:
    with sf.SoundFile(args.filename, mode='x', samplerate=args.samplerate,
                      channels=args.channels, subtype=args.subtype) as file:
        with sd.InputStream(samplerate=args.samplerate, device=args.device,
                            channels=args.channels, callback=callback):
            print('#' * 80)
            print('press Ctrl+C to stop the recording')
            print('#' * 80)
            while True:
                file.write(q.get())
except KeyboardInterrupt:
    print('\nRecording finished: ' + repr(args.filename))
    parser.exit(0)
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

3.10 Using a stream in an asyncio[26] coroutine

asyncio_coroutines.py[27]

```
#!/usr/bin/env python3
"""An example for using a stream in an asyncio coroutine.

This example shows how to create a stream in a coroutine and how to wait for
the completion of the stream.

You need Python 3.7 or newer to run this.

"""
import asyncio
import sys

import numpy as np
import sounddevice as sd


async def record_buffer(buffer, **kwargs):
    loop = asyncio.get_event_loop()
    event = asyncio.Event()
    idx = 0

    def callback(indata, frame_count, time_info, status):
        nonlocal idx
```

(continues on next page)

[26] https://docs.python.org/3/library/asyncio.html#module-asyncio
[27] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/asyncio_coroutines.py

(continued from previous page)

```
        if status:
            print(status)
        remainder = len(buffer) - idx
        if remainder == 0:
            loop.call_soon_threadsafe(event.set)
            raise sd.CallbackStop
        indata = indata[:remainder]
        buffer[idx:idx + len(indata)] = indata
        idx += len(indata)

    stream = sd.InputStream(callback=callback, dtype=buffer.dtype,
                            channels=buffer.shape[1], **kwargs)
    with stream:
        await event.wait()


async def play_buffer(buffer, **kwargs):
    loop = asyncio.get_event_loop()
    event = asyncio.Event()
    idx = 0

    def callback(outdata, frame_count, time_info, status):
        nonlocal idx
        if status:
            print(status)
        remainder = len(buffer) - idx
        if remainder == 0:
            loop.call_soon_threadsafe(event.set)
            raise sd.CallbackStop
        valid_frames = frame_count if remainder >= frame_count else remainder
        outdata[:valid_frames] = buffer[idx:idx + valid_frames]
        outdata[valid_frames:] = 0
        idx += valid_frames

    stream = sd.OutputStream(callback=callback, dtype=buffer.dtype,
                             channels=buffer.shape[1], **kwargs)
    with stream:
        await event.wait()


async def main(frames=150_000, channels=1, dtype='float32', **kwargs):
    buffer = np.empty((frames, channels), dtype=dtype)
    print('recording buffer ...')
    await record_buffer(buffer, **kwargs)
    print('playing buffer ...')
    await play_buffer(buffer, **kwargs)
    print('done')


if __name__ == "__main__":
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        sys.exit('\nInterrupted by user')
```

3.11 Creating an asyncio[28] generator for audio blocks

asyncio_generators.py[29]

```
#!/usr/bin/env python3
"""Creating an asyncio generator for blocks of audio data.

This example shows how a generator can be used to analyze audio input blocks.
In addition, it shows how a generator can be created that yields not only input
blocks but also output blocks where audio data can be written to.

You need Python 3.7 or newer to run this.

"""
import asyncio
import queue
import sys

import numpy as np
import sounddevice as sd


async def inputstream_generator(channels=1, **kwargs):
    """Generator that yields blocks of input data as NumPy arrays."""
    q_in = asyncio.Queue()
    loop = asyncio.get_event_loop()

    def callback(indata, frame_count, time_info, status):
        loop.call_soon_threadsafe(q_in.put_nowait, (indata.copy(), status))

    stream = sd.InputStream(callback=callback, channels=channels, **kwargs)
    with stream:
        while True:
            indata, status = await q_in.get()
            yield indata, status


async def stream_generator(blocksize, *, channels=1, dtype='float32',
                           pre_fill_blocks=10, **kwargs):
    """Generator that yields blocks of input/output data as NumPy arrays.

    The output blocks are uninitialized and have to be filled with
    appropriate audio signals.

    """
    assert blocksize != 0
    q_in = asyncio.Queue()
    q_out = queue.Queue()
    loop = asyncio.get_event_loop()

    def callback(indata, outdata, frame_count, time_info, status):
        loop.call_soon_threadsafe(q_in.put_nowait, (indata.copy(), status))
        outdata[:] = q_out.get_nowait()

    # pre-fill output queue
    for _ in range(pre_fill_blocks):
```

(continues on next page)

[28] https://docs.python.org/3/library/asyncio.html#module-asyncio
[29] https://github.com/spatialaudio/python-sounddevice/blob/0.5.2/examples/asyncio_generators.py

(continued from previous page)

```
        q_out.put(np.zeros((blocksize, channels), dtype=dtype))

    stream = sd.Stream(blocksize=blocksize, callback=callback, dtype=dtype,
                       channels=channels, **kwargs)
    with stream:
        while True:
            indata, status = await q_in.get()
            outdata = np.empty((blocksize, channels), dtype=dtype)
            yield indata, outdata, status
            q_out.put_nowait(outdata)


async def print_input_infos(**kwargs):
    """Show minimum and maximum value of each incoming audio block."""
    async for indata, status in inputstream_generator(**kwargs):
        if status:
            print(status)
        print('min:', indata.min(), '\t', 'max:', indata.max())


async def wire_coro(**kwargs):
    """Create a connection between audio inputs and outputs.

    Asynchronously iterates over a stream generator and for each block
    simply copies the input data into the output block.

    """
    async for indata, outdata, status in stream_generator(**kwargs):
        if status:
            print(status)
        outdata[:] = indata


async def main(**kwargs):
    print('Some information about the input signal:')
    try:
        await asyncio.wait_for(print_input_infos(), timeout=2)
    except asyncio.TimeoutError:
        pass
    print('\nEnough of that, activating wire ...\n')
    audio_task = asyncio.create_task(wire_coro(**kwargs))
    for i in range(10, 0, -1):
        print(i)
        await asyncio.sleep(1)
    audio_task.cancel()
    try:
        await audio_task
    except asyncio.CancelledError:
        print('\nwire was cancelled')


if __name__ == "__main__":
    try:
        asyncio.run(main(blocksize=1024))
    except KeyboardInterrupt:
        sys.exit('\nInterrupted by user')
```

4 Contributing

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

4.1 Reporting Problems

When creating an issue at https://github.com/spatialaudio/python-sounddevice/issues, please make sure to provide as much useful information as possible.

You can use Markdown formatting to show Python code, e.g.

````
I have created a script named `my_script.py`:

```python
import sounddevice as sd

fs = 48000
duration = 1.5

data = sd.rec(int(duration * fs), channels=99)
sd.wait()
print(data.shape)
```
````

Please provide minimal code (remove everything that's not necessary to show the problem), but make sure that the code example still has everything that's needed to run it, including all `import` statements.

You should of course also show what happens when you run your code, e.g.

````
Running my script, I got this error:

```
$ python my_script.py
Expression 'parameters->channelCount <= maxChans' failed in 'src/hostapi/alsa/pa_
↪linux_alsa.c', line: 1514
Expression 'ValidateParameters( inputParameters, hostApi, StreamDirection_In )'␣
↪failed in 'src/hostapi/alsa/pa_linux_alsa.c', line: 2818
Traceback (most recent call last):
  File "my_script.py", line 6, in <module>
    data = sd.rec(int(duration * fs), channels=99)
...
sounddevice.PortAudioError: Error opening InputStream: Invalid number of channels␣
↪[PaErrorCode -9998]
```
````

Please remember to provide the full command invocation and the full output. You should only remove lines of output when you know they are irrelevant.

You should also mention the operating system and host API you are using (e.g. "Linux/ALSA" or "macOS/Core Audio" or "Windows/WASAPI").

If your problem is related to a certain hardware device, you should provide the list of devices as reported by

```
python -m sounddevice
```

If your problem has to do with the version of the PortAudio library you are using, you should provide the output of this script:

```
import sounddevice as sd
print(sd._libname)
print(sd.get_portaudio_version())
```

If you don't want to clutter the issue description with a huge load of gibberish, you can use the `<details>` HTML tag to show some content only on demand:

````
<details>

```
$ python -m sounddevice
  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)
```

</details>
````

4.2 Development Installation

Instead of pip-installing the latest release from PyPI[30], you should get the newest development version (a.k.a. "master") from Github[31]:

```
git clone --recursive https://github.com/spatialaudio/python-sounddevice.git
cd python-sounddevice
python -m pip install -e .
```

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

Whenever the file `sounddevice_build.py` changes (either because you edited it or it was updated by pulling from Github or switching branches), you have to run the last command again.

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

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

4.3 Building the Documentation

If you make changes to the documentation, you can locally re-create the HTML pages using Sphinx[32]. You can install it and a few other necessary packages with:

```
python -m pip install -r doc/requirements.txt
```

To (re-)build the HTML files, use:

```
python -m sphinx doc _build
```

The generated files will be available in the directory `_build/`.

If you additionally install sphinx-autobuild[33], you can run Sphinx automatically on any changes and conveniently auto-reload the changed pages in your browser:

[30] https://pypi.org/project/sounddevice/
[31] https://github.com/spatialaudio/python-sounddevice/
[32] https://www.sphinx-doc.org/
[33] https://github.com/executablebooks/sphinx-autobuild

```
python -m sphinx_autobuild doc _build --open-browser
```

5 API Documentation

Play and Record Sound with Python.

API overview:

- Convenience functions to play and record NumPy arrays: *play()*, *rec()*, *playrec()* and the related functions *wait()*, *stop()*, *get_status()*, *get_stream()*
- Functions to get information about the available hardware: *query_devices()*, *query_hostapis()*, *check_input_settings()*, *check_output_settings()*
- Module-wide default settings: *default*
- Platform-specific settings: *AsioSettings*, *CoreAudioSettings*, *WasapiSettings*
- PortAudio streams, using NumPy arrays: *Stream*, *InputStream*, *OutputStream*
- PortAudio streams, using Python buffer objects (NumPy not needed): *RawStream*, *RawInputStream*, *RawOutputStream*
- Miscellaneous functions and classes: *sleep()*, *get_portaudio_version()*, *CallbackFlags*, *CallbackStop*, *CallbackAbort*

Online documentation:
https://python-sounddevice.readthedocs.io/

5.1 Convenience Functions using NumPy Arrays

Overview

play	Play back a NumPy array containing audio data.
rec	Record audio data into a NumPy array.
playrec	Simultaneous playback and recording of NumPy arrays.
wait	Wait for *play()*/*rec()*/*playrec()* to be finished.
stop	Stop playback/recording.
get_status	Get info about over-/underflows in *play()*/*rec()*/*playrec()*.
get_stream	Get a reference to the current stream.

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

Play back a NumPy array containing audio data.

This is a convenience function for interactive use and for small scripts. It cannot be used for multiple overlapping playbacks.

This function does the following steps internally:

- Call *stop()* to terminate any currently running invocation of *play()*, *rec()* and *playrec()*.
- Create an *OutputStream* and a callback function for taking care of the actual playback.
- Start the stream.
- If `blocking=True` was given, wait until playback is done. If not, return immediately (to start waiting at a later point, *wait()* can be used).

If you need more control (e.g. block-wise gapless playback, multiple overlapping playbacks, ...), you should explicitly create an *OutputStream* yourself. If NumPy is not available, you can use a *RawOutputStream*.

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 simply 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, in which case the signal is played back on all given output channels). Each channel number may only appear once in *mapping*.
- **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()*.
- **loop** (*bool, optional*) – Play *data* in a loop.

Other Parameters

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

Notes

If you don't specify the correct sampling rate (either with the *samplerate* argument or by assigning a value to *default.samplerate*), the audio data will be played back, but it might be too slow or too fast!

See also

rec, *playrec*

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

Record audio data into a NumPy array.

This is a convenience function for interactive use and for small scripts.

This function does the following steps internally:

- Call *stop()* to terminate any currently running invocation of *play()*, *rec()* and *playrec()*.
- Create an *InputStream* and a callback function for taking care of the actual recording.
- Start the stream.
- If `blocking=True` was given, wait until recording is done. If not, return immediately (to start waiting at a later point, *wait()* can be used).

If you need more control (e.g. block-wise gapless recording, overlapping recordings, ...), you should explicitly create an *InputStream* yourself. If NumPy is not available, you can use a *RawInputStream*.

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.

Notes

If you don't specify a sampling rate (either with the *samplerate* argument or by assigning a value to *default.samplerate*), the default sampling rate of the sound device will be used (see *query_devices()*).

> **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 of NumPy arrays.

This function does the following steps internally:

- Call *stop()* to terminate any currently running invocation of *play()*, *rec()* and *playrec()*.
- Create a *Stream* and a callback function for taking care of the actual playback and recording.
- Start the stream.
- If `blocking=True` was given, wait until playback/recording is done. If not, return immediately (to start waiting at a later point, *wait()* can be used).

If you need more control (e.g. block-wise gapless playback and recording, realtime processing, ...), you should explicitly create a *Stream* yourself. If NumPy is not available, you can use a *RawStream*.

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.

Notes

If you don't specify the correct sampling rate (either with the *samplerate* argument or by assigning a value to `default.samplerate`), the audio data will be played back, but it might be too slow or too fast!

See also

`play`, `rec`

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

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

Playback/recording can be stopped with a `KeyboardInterrupt`[34].

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 info 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`

[34] https://docs.python.org/3/library/exceptions.html#KeyboardInterrupt

sounddevice.**get_stream**()

Get a reference to the current stream.

This applies only to streams created by calls to *play()*, *rec()* or *playrec()*.

Returns

Stream – An *OutputStream*, *InputStream* or *Stream* associated with the last invocation of *play()*, *rec()* or *playrec()*, respectively.

5.2 Checking Available Hardware

Overview

query_devices	Return information about available devices.
DeviceList	A list with information about all available audio devices.
query_hostapis	Return information about available host APIs.
check_input_settings	Check if given input device settings are supported.
check_output_settings	Check if given output device settings are supported.

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(s), 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.
- **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 arguments were specified – a *DeviceList* containing one dictionary for each available device. The dictionaries have the following keys:

`'name'`
The name of the device.

`'index'`
The device index.

`'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:

```
python3 -m sounddevice
```

Examples

The returned *DeviceList* can be indexed and iterated over like any sequence type (yielding the abovementioned 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)
   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 macOS, 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)
```

class sounddevice.**DeviceList**(*iterable=()*, */ (Positional-only parameter separator (PEP 570))*)

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()`[35] function. Furthermore, it can be obtained with `repr()`[36] and `str()`[37].

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*, *extra_settings=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(s), 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*.
- **extra_settings** (*settings object, optional*) – This can be used for host-API-specific input settings. See *default.extra_settings*.
- **samplerate** (*float, optional*) – Sampling frequency, see *default.samplerate*.

[35] https://docs.python.org/3/library/functions.html#print
[36] https://docs.python.org/3/library/functions.html#repr
[37] https://docs.python.org/3/library/stdtypes.html#str

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

Check if given output device settings are supported.

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

5.3 Module-wide Default Settings

class sounddevice.**default**

Get/set defaults for the *sounddevice* module.

The attributes *device*, *channels*, *dtype*, *latency* and *extra_settings* 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.

See the *device* argument of *Stream*.

If not overwritten, this is queried from PortAudio.

> **See also**
>
> *query_devices()*

channels = (None, None)

Default number of input/output channels.

See the *channels* argument of *Stream*.

> **See also**
>
> *query_devices()*

dtype = ('float32', 'float32')

Default data type used for input/output samples.

See the *dtype* argument of *Stream*.

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, which is *not* supported in NumPy!).

latency = ('high', 'high')

See the *latency* argument of *Stream*.

extra_settings = (None, None)

Host-API-specific input/output settings.

> **See also**
>
> *AsioSettings*, *CoreAudioSettings*, *WasapiSettings*

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. See also http://www.portaudio.com/docs/proposals/001-UnderflowOverflowHandling.html.

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). See also http://www.portaudio.com/docs/proposals/020-AllowCallbackToPrimeStream.html.

property hostapi

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

reset()

Reset all attributes to their "factory default".

5.4 Platform-specific Settings

Overview

AsioSettings	ASIO-specific input/output settings.
CoreAudioSettings	Mac Core Audio-specific input/output settings.
WasapiSettings	WASAPI-specific input/output settings.

class sounddevice.**AsioSettings**(*channel_selectors*)

ASIO-specific input/output settings.

Objects of this class can be used as *extra_settings* argument to *Stream()* (and variants) or as *default.extra_settings*.

Parameters

channel_selectors (*list of int*) – Support for opening only specific channels of an ASIO device. *channel_selectors* is a list of integers specifying the (zero-based) channel numbers to use. The length of *channel_selectors* must match the corresponding *channels* parameter of *Stream()* (or variants), otherwise a crash may result. The values in the *channel_selectors* array must specify channels within the range of supported channels.

Examples

Setting output channels when calling *play()*:

```
>>> import sounddevice as sd
>>> asio_out = sd.AsioSettings(channel_selectors=[12, 13])
>>> sd.play(..., extra_settings=asio_out)
```

Setting default output channels:

```
>>> sd.default.extra_settings = asio_out
>>> sd.play(...)
```

Setting input channels as well:

```
>>> asio_in = sd.AsioSettings(channel_selectors=[8])
>>> sd.default.extra_settings = asio_in, asio_out
>>> sd.playrec(..., channels=1, ...)
```

class sounddevice.**CoreAudioSettings**(*channel_map=None*, *change_device_parameters=False*, *fail_if_conversion_required=False*, *conversion_quality='max'*)

Mac Core Audio-specific input/output settings.

Objects of this class can be used as *extra_settings* argument to *Stream()* (and variants) or as *default.extra_settings*.

Parameters

- **channel_map** (*sequence of int, optional*) – Support for opening only specific channels of a Core Audio device. Note that *channel_map* is treated differently between input and output channels.

  For input devices, *channel_map* is a list of integers specifying the (zero-based) channel numbers to use.

  For output devices, *channel_map* must have the same length as the number of output channels of the device. Specify unused channels with -1, and a 0-based index for any desired channels.

  See the example below. For additional information, see the PortAudio documentation[38].

- **change_device_parameters** (*bool, optional*) – If `True`, allows PortAudio to change things like the device's frame size, which allows for much lower latency, but might disrupt the device if other programs are using it, even when you are just querying the device. `False` is the default.

- **fail_if_conversion_required** (*bool, optional*) – In combination with the above flag, `True` causes the stream opening to fail, unless the exact sample rates are supported by the device.

- **conversion_quality** (*{'min', 'low', 'medium', 'high', 'max'}, optional*) – This sets Core Audio's sample rate conversion quality. `'max'` is the default.

Example

This example assumes a device having 6 input and 6 output channels. Input is from the second and fourth channels, and output is to the device's third and fifth channels:

```
>>> import sounddevice as sd
>>> ca_in = sd.CoreAudioSettings(channel_map=[1, 3])
>>> ca_out = sd.CoreAudioSettings(channel_map=[-1, -1, 0, -1, 1, -1])
>>> sd.playrec(..., channels=2, extra_settings=(ca_in, ca_out))
```

class sounddevice.**WasapiSettings**(*exclusive=False*, *auto_convert=False*)

WASAPI-specific input/output settings.

Objects of this class can be used as *extra_settings* argument to *Stream()* (and variants) or as *default.extra_settings*. They can also be used in *check_input_settings()* and *check_output_settings()*.

Parameters

- **exclusive** (*bool*) – Exclusive mode allows to deliver audio data directly to hardware bypassing software mixing.

- **auto_convert** (*bool*) – Allow WASAPI backend to insert system-level channel matrix mixer and sample rate converter to support playback formats that do not match the current configured system settings. This is in particular required for streams not matching the system mixer sample rate. This only applies in *shared mode* and has no effect when *exclusive* is set to `True`.

Examples

Setting exclusive mode when calling *play()*:

```
>>> import sounddevice as sd
>>> wasapi_exclusive = sd.WasapiSettings(exclusive=True)
>>> sd.play(..., extra_settings=wasapi_exclusive)
```

Setting exclusive mode as default:

[38] https://app.assembla.com/spaces/portaudio/git/source/master/src/hostapi/coreaudio/notes.txt

```
>>> sd.default.extra_settings = wasapi_exclusive
>>> sd.play(...)
```

5.5 Streams using NumPy Arrays

Overview

Stream	PortAudio stream for simultaneous input and output (using NumPy).
InputStream	PortAudio input stream (using NumPy).
OutputStream	PortAudio output stream (using NumPy).

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

PortAudio stream for simultaneous input and output (using NumPy).

To open an input-only or output-only stream use *InputStream* or *OutputStream*, respectively. If you want to handle audio data as plain 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[39], i.e. it can be used in a with statement[40] 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 min-

imum 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*. 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.
- **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'` or `'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 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[41] provided to the stream callback, *read()* or *write()*. It may be any of *float32*, *int32*, *int16*, *int8*, *uint8*. See numpy.dtype[42]. 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*. If NumPy is available, the corresponding numpy.dtype[43] 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** (*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 device's default low and high latency, respectively (see *query_devices()*). `'high'` is typically more robust (i.e. buffer under-/overflows are less likely), but the latency may be too large for interactive applications.

  Note

  Specifying the desired latency as `'high'` does not *guarantee* a stable audio stream. For reference, by default Audacity[44] specifies a desired latency of `0.1` seconds and typically achieves robust performance.

  The default value(s) can be changed with *default.latency*. Actual latency values for an open stream can be retrieved using the *latency* attribute.
- **extra_settings** (*settings object or pair thereof, optional*) – This can be used for host-API-specific input/output settings. See *default.extra_settings*.
- **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`[45] 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.

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`[46]. 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*.
- **prime_output_buffers_using_stream_callback** (*bool, optional*) – See *default.prime_output_buffers_using_stream_callback*.

abort(*ignore_errors=True*)

Terminate audio processing immediately.

This does not wait for pending buffers to complete.

> **See also**
>
> *start*, *stop*

property 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*

property blocksize

Number of frames per block.

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

property 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.

property closed

`True` after a call to *close()*, `False` otherwise.

property 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.

property device

IDs of the input/output device.

property dtype

Data type of the audio samples.

> **See also**
>
> *default.dtype*, *samplesize*

property 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`[47] 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.

property 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.

property 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()*.

property samplesize

The size in bytes of a single sample.

> **See also**
>
> *dtype*

start()

Commence audio processing.

> **See also**
>
> *stop*, *abort*

stop(*ignore_errors=True*)

Terminate audio processing.

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

> **See also**
>
> *start*, *abort*

property 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*

property 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()`[48]).

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.

property 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*, *extra_settings=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)

PortAudio input stream (using NumPy).

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*

[39] https://docs.python.org/3/reference/datamodel.html#context-managers
[40] https://docs.python.org/3/reference/compound_stmts.html#with
[41] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[42] https://numpy.org/doc/stable/reference/generated/numpy.dtype.html#numpy.dtype
[43] https://numpy.org/doc/stable/reference/generated/numpy.dtype.html#numpy.dtype
[44] https://www.audacityteam.org/
[45] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[46] https://docs.python.org/3/library/sys.html#sys.stderr
[47] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[48] https://numpy.org/doc/stable/reference/generated/numpy.ascontiguousarray.html#numpy.ascontiguousarray

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

PortAudio output stream (using NumPy).

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*

5.6 Raw Streams

Overview

RawStream	PortAudio input/output stream (using buffer objects).
RawInputStream	PortAudio input stream (using buffer objects).
RawOutputStream	PortAudio output stream (using buffer objects).

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

PortAudio input/output stream (using buffer objects).

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 for using this.

To open a "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 *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 for using 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 *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 for using 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 *samplesize*.

Returns

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

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

PortAudio input stream (using buffer objects).

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 for using 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*, *extra_settings=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*)

PortAudio output stream (using buffer objects).

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 for using 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*

5.7 Miscellaneous

Overview

sleep	Put the caller to sleep for at least *msec* milliseconds.
get_portaudio_version	Get version information for the PortAudio library.
CallbackFlags	Flag bits for the *status* argument to a stream *callback*.
CallbackStop	Exception to be raised by the user to stop callback processing.
CallbackAbort	Exception to be raised by the user to abort callback processing.
PortAudioError	This exception will be raised on PortAudio errors.

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.**CallbackFlags**(*flags=0*)

Flag bits for the *status* argument to a stream *callback*.

If you experience under-/overflows, you can try to increase the `latency` and/or `blocksize` settings. You should also avoid anything that could block the callback function for a long time, e.g. extensive computations, waiting for another thread, reading/writing files, network connections, etc.

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
```

The values may also be set and cleared by the user:

```
>>> import sounddevice as sd
>>> cf = sd.CallbackFlags()
>>> cf
<sounddevice.CallbackFlags: no flags set>
>>> cf.input_underflow = True
>>> cf
<sounddevice.CallbackFlags: input underflow>
>>> cf.input_underflow = False
>>> cf
<sounddevice.CallbackFlags: no flags set>
```

property 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.

This can only happen in full-duplex streams (including *playrec()*).

property 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. In a stream opened with a non-zero *blocksize*, it indicates that data prior to one or more samples in the input buffer was discarded.

This can happen in full-duplex and input-only streams (including *playrec()* and *rec()*).

property output_underflow

Output underflow.

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

This can happen in full-duplex and output-only streams (including *playrec()* and *play()*).

property output_overflow

Output overflow.

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

This can only happen in full-duplex streams (including *playrec()*), but only when `never_drop_input=True` was specified. See *default.never_drop_input*.

property priming_output

Priming output.

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

This will only take place with some of the host APIs, and only if `prime_output_buffers_using_stream_callback=True` was specified. See *default.prime_output_buffers_using_stream_callback*.

exception 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

exception 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

exception sounddevice.**PortAudioError**

This exception will be raised on PortAudio errors.

args

A variable length tuple containing the following elements when available:

1) A string describing the error
2) The PortAudio `PaErrorCode` value
3) A 3-tuple containing the host API index, host error code, and the host error message (which may be an empty string)

5.8 Expert Mode

Overview

_initialize	Initialize PortAudio.
_terminate	Terminate PortAudio.
_split	Split input/output value into two values.
_StreamBase	Base class for PortAudio streams.

sounddevice.**_initialize**()

Initialize PortAudio.

This temporarily forwards messages from stderr to `/dev/null` (where supported).

In most cases, this doesn't have to be called explicitly, because it is automatically called with the `import sounddevice` statement.

sounddevice.**_terminate**()

Terminate PortAudio.

In most cases, this doesn't have to be called explicitly.

sounddevice.**_split**(*value*)

Split input/output value into two values.

This can be useful for generic code that allows using the same value for input and output but also a pair of two separate values.

class sounddevice.**_StreamBase**(*kind*, *samplerate=None*, *blocksize=None*, *device=None*, *channels=None*, *dtype=None*, *latency=None*, *extra_settings=None*, *callback=None*, *finished_callback=None*, *clip_off=None*, *dither_off=None*, *never_drop_input=None*, *prime_output_buffers_using_stream_callback=None*, *userdata=None*, *wrap_callback=None*)

Base class for PortAudio streams.

This class should only be used by library authors who want to create their own custom stream classes. Most users should use the derived classes *Stream*, *InputStream*, *OutputStream*, *RawStream*, *RawInputStream* and *RawOutputStream* instead.

This class has the same properties and methods as *Stream*, except for *read_available*/*read()* and *write_available*/*write()*.

It can be created with the same parameters as *Stream*, except that there are three additional parameters and the *callback* parameter also accepts a C function pointer.

Parameters

- **kind** (*{'input', 'output', 'duplex'}*) – The desired type of stream: for recording, playback or both.
- **callback** (*Python callable or CData function pointer, optional*) – If *wrap_callback* is `None` this can be a function pointer provided by CFFI. Otherwise, it has to be a Python callable.
- **wrap_callback** (*{'array', 'buffer'}, optional*) – If *callback* is a Python callable, this selects whether the audio data is provided as NumPy array (like in *Stream*) or as Python buffer object (like in *RawStream*).
- **userdata** (*CData void pointer*) – This is passed to the underlying C callback function on each call and can only be accessed from a *callback* provided as `CData` function pointer.

Examples

A usage example of this class can be seen at https://github.com/spatialaudio/python-rtmixer.

6 Version History

0.5.2 (2025-05-16):

- Better error if frames/channels are non-integers

0.5.1 (2024-10-12):

- Windows wheel: bundle both non-ASIO and ASIO DLLs, the latter can be chosen by defining the `SD_ENABLE_ASIO` environment variable

0.5.0 (2024-08-11):

- Remove ASIO support from bundled DLLs (DLLs with ASIO can be manually selected)

0.4.7 (2024-05-27):

- support `paWinWasapiAutoConvert` with `auto_convert` flag in *`WasapiSettings`*
- Avoid exception in *`PortAudioError`*`.__str__()`

0.4.6 (2023-02-19):

- Redirect stderr with `os.dup2()` instead of CFFI calls

0.4.5 (2022-08-21):

- Add `index` field to device dict
- Require Python >= 3.7
- Add PaWasapi_IsLoopback() to cdef (high-level interface not yet available)

0.4.4 (2021-12-31):

- Exact device string matches can now include the host API name

0.4.3 (2021-10-20):

- Fix dimension check in *`Stream.write()`*
- Provide "universal" (x86_64 and arm64) `.dylib` for macOS

0.4.2 (2021-07-18):

- Update PortAudio binaries to version 19.7.0
- Wheel names are now shorter

0.4.1 (2020-09-26):

- *`CallbackFlags`* attributes are now writable

0.4.0 (2020-07-18):

- Drop support for Python 2.x
- Fix memory issues in *`play()`*, *`rec()`* and *`playrec()`*
- Example application `play_stream.py`

0.3.15 (2020-03-18):

- This will be the last release supporting Python 2.x!

0.3.14 (2019-09-25):

- Examples `play_sine.py` and `rec_gui.py`
- Redirect `stderr` only during initialization

0.3.13 (2019-02-27):

- Examples `asyncio_coroutines.py` and `asyncio_generators.py`

0.3.12 (2018-09-02):

- Support for the dylib from Anaconda

0.3.11 (2018-05-07):

- Support for the DLL from `conda-forge`

0.3.10 (2017-12-22):

- Change the way how the PortAudio library is located

0.3.9 (2017-10-25):

- Add *Stream.closed*
- Switch CFFI usage to "out-of-line ABI" mode

0.3.8 (2017-07-11):

- Add more `ignore_errors` arguments
- Add *PortAudioError.args*
- Add *CoreAudioSettings*

0.3.7 (2017-02-16):

- Add *get_stream()*
- Support for CData function pointers as callbacks

0.3.6 (2016-12-19):

- Example application `play_long_file.py`

0.3.5 (2016-09-12):

- Add `extra_settings` option for host-API-specific stream settings
- Add *AsioSettings* and *WasapiSettings*

0.3.4 (2016-08-05):

- Example application `rec_unlimited.py`

0.3.3 (2016-04-11):

- Add `loop` argument to *play()*

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 *query_devices()*

0.3.1 (2016-01-04):

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

0.3.0 (2015-10-28):

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

0.2.2 (2015-10-21):

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

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)

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 the soundfile[49] module.
- Remove logging (this seemed too intrusive)
- Return callback status from *`wait()`* and add the new function *`get_status()`*
- *`playrec()`*: Rename the arguments *input_channels* and *input_dtype* to *channels* and *dtype*, respectively

0.1.0 (2015-06-20):

Initial release. Some ideas are taken from PySoundCard[50]. Thanks to Bastian Bechtold for many fruitful discussions during the development of several features which *python-sounddevice* inherited from there.

[49] https://github.com/bastibe/python-soundfile/
[50] https://github.com/bastibe/PySoundCard/