audiomath Documentation

Release 1.15.2

Jeremy Hill

Feb 18, 2021

Table of Contents

1 Installation **3**
- 1.1 Compatibility and requirements . . . 3
- 1.2 Normal installation . . . 3
- 1.3 Advanced installation (from version-controlled sources) . . . 4

2 How to... **5**
- 2.1 Read a sound from a file into memory . . . 5
- 2.2 Write a sound from memory into a file . . . 5
- 2.3 Create a sound from scratch, in memory . . . 6
- 2.4 Define a functionally-generated sound . . . 6
- 2.5 Perform simple arithmetic . . . 7
- 2.6 Extract a segment of a sound . . . 7
- 2.7 Extract selected channels from multi-channel sounds . . . 8
- 2.8 Concatenate sounds in time (splicing) . . . 8
- 2.9 Stack channels (multiplexing) . . . 9
- 2.10 Pitch-shift or time-stretch a sound . . . 10
- 2.11 Preprocess a sound using SoX . . . 10
- 2.12 Manipulate a sound in other miscellaneous ways . . . 10
- 2.13 Plot a sound . . . 11
- 2.14 Play sounds . . . 11
- 2.15 Play sounds with more-precise latency, via `PsychPortAudio` . . . 14
- 2.16 Record a sound into memory . . . 15
- 2.17 Record a sound directly to file . . . 15
- 2.18 Target a particular input or output device . . . 15
- 2.19 Manipulate the operating-system's overall volume settings . . . 17
- 2.20 Measure audio latency . . . 17

3 API reference **19**
- 3.1 The `audiomath` package . . . 19
  - 3.1.1 Summary . . . 19
  - 3.1.2 Classes . . . 19
  - 3.1.3 Global functions and constants . . . 20
- 3.2 The `Sound` class . . . 22
- 3.3 The `Synth` class . . . 30
- 3.4 The `Player` class . . . 31
- 3.5 The `Queue` class . . . 38
- 3.6 The `Fader` class . . . 40

3.7 The Delay class . . . 41
3.8 The Recorder class . . . 41
3.9 The ffmpeg class . . . 44
3.10 The sox class . . . 46
3.11 The audiomath.PortAudioInterface sub-module . . . 48
3.12 The audiomath.PsychToolboxInterface sub-module . . . 52
3.13 The audiomath.Signal sub-module . . . 56
3.14 The audiomath.SystemVolume sub-module . . . 60

4 Support **63**

5 License **65**

6 Indices and tables **67**

Python Module Index **69**

Index **71**

- Permanent home: https://pypi.org/project/audiomath
- Documentation: https://audiomath.readthedocs.io
- Source code: https://bitbucket.org/snapproject/audiomath-gitrepo

When reporting any work in which you used *audiomath*, please cite:

- Hill NJ, Mooney SWJ & Prusky GT (2021). audiomath: a neuroscientist's sound toolkit. Heliyon 7(2):e06236. https://doi.org/10.1016/j.heliyon.2021.e06236

```
@article{hill2020audiomath,
    author  = {Hill, N. Jeremy  and  Mooney, Scott W. J.  and  Prusky, Glen T.},
    title   = {{audiomath}: a Neuroscientist's Sound Toolkit},
    journal = {Heliyon},
    volume  = {7},
    number  = {2},
    pages   = {e06236},
    month   = {February},
    year    = {2021},
    date    = {2021-02-10},
    doi     = {10.1016/j.heliyon.2021.e06236},
    url     = {https://doi.org/10.1016/j.heliyon.2021.e06236},
}
```

audiomath is a package for Python programmers who want to record, manipulate, visualize or play sound waveforms. It allows you to:

- Represent sounds as numeric arrays via the third-party Python package `numpy` (required). The arrays are contained within high-level objects, allowing common operations to be performed with minimal coding—for example: slicing and concatenation in time, selection and stacking of channels, resampling, mixing, rescaling and modulation.
- Plot the resulting waveforms, via the third-party Python package `matplotlib` (optional).
- Read and write uncompressed `wav` files (via the Python standard library).
- Read other audio formats using the third-party AVbin library (binaries are included in the package, for a selection of platforms).
- Record and play back sounds using the third-party PortAudio library (binaries are included in the package, for a selection of platforms).
- Play sounds using the alternative PsychPortAudio back-end, from the (optional) `psychtoolbox` package, allowing very precise control of latency
- Plug in other recording/playback back-ends with moderate development effort.

CHAPTER 1

Installation

1.1 Compatibility and requirements

audiomath is compatible with Python 2.7, and with Python 3.3+.

For your own Python development, it is highly advisable to install a Python distribution that you keep separate from whatever distribution came with your operating system. Anaconda is a good choice for this. At any rate, we'll assume that when we direct you to type `python` or `pip` in your Terminal app or Command Prompt, you have figured out how to ensure that the the correct distribution of Python is being addressed by these commands. On Windows, the "Anaconda Prompt" is a good shortcut to use to ensure this.

audiomath requires the third-party package `numpy` for almost everything it does. When you use `pip` to install *audiomath*, it will ensure `numpy` is installed too. On Windows, it will also install `comtypes` and `psutil` since these are used by the *audiomath.SystemVolume* submodule.

If you want to plot sound waveforms, you will also need the third-party package `matplotlib`. This is optional, so it will be left to you to `python -m pip install matplotlib` if you want it and do not already have it.

Similarly, you may want to install the third-party package `librosa` if you want to time-stretch or pitch-shift sounds.

1.2 Normal installation

To download the latest release from pypi.org and install it into your Python distribution:

```
python -m pip install audiomath
```

Later, when you want to upgrade an existing installation to the latest, greatest version:

```
python -m pip install --upgrade audiomath --no-dependencies
```

1.3 Advanced installation (from version-controlled sources)

To work with the latest sources from the git repository, you will need to ensure the `git` command-line tool is installed, and then:

```
cd WHEREVER-YOU-WOULD-LIKE-TO-KEEP-AUDIOMATH-LONG-TERM
git clone https://bitbucket.org/snapproject/audiomath-gitrepo
cd audiomath-gitrepo
git checkout origin/release --track   # creates local branch called `release`
git checkout master   # back to master (unless you want to stay on `release`)
python -m pip install -e .
```

Note the `-e` flag, which tells Python to install the repository as an "editable" package. The repository's `master` branch is the bleeding edge, whereas the `release` branch replicates the pypi.org releases. Use `git checkout master` or `git checkout release` to switch between them. In either case, when you want to upgrade to the latest version from the server later on:

```
cd audiomath-gitrepo
git fetch
git checkout master  && git merge   # update to the latest changes in the master␣
↪branch
git checkout release && git merge   # update to the latest changes in the release␣
↪branch
git checkout master   # (assuming you want to be working on master, go back there)
```

CHAPTER 2

How to...

This page provides a tour of *audiomath*'s main features. The examples all use the following conventions:

- `am` is an alias for the *audiomath* package, created with `import audiomath as am`;
- `s` is a `Sound` instance (as are `s1`, `s2`, etc.);
- `p` is a `Player` instance;
- `r` is a `Recorder` instance.

Many of the examples can be tried out easily using the built-in test sound:

```
s = am.TestSound()       # returns an 8-channel Sound
s = am.TestSound('12')   # returns just the first two channels
```

2.1 Read a sound from a file into memory

```
s = am.Sound('some_file.wav')   # uncompressed .wav files can be read using
                                # the standard Python library, under the hood

s = am.Sound('some_file.mp3')   # other formats require the third-party AVBin
                                # library---AVBin binaries for the most common
                                # OS platforms are included within audiomath
```

2.2 Write a sound from memory into a file

```
s.Write('some_file.wav')        # uncompressed .wav files can be written using
                                # the standard Python library, under the hood

s.Write('some_file.flac')       # other formats require the third-party
```

(continues on next page)

(continued from previous page)

```
                                    # command-line utility `ffmpeg` or `sox` to be
                                    # installed on your system (you'll need to do
                                    # this yourself---see the doc for the `am.ffmpeg`
                                    # and `am.sox` classes for more info)
```

2.3 Create a sound from scratch, in memory

A specified number of seconds of silence:

```
s = am.Sound(5)                             # five seconds of silence
s = am.Sound(5, fs=44100, nChannels=2)      # the same, with sampling frequency and
                                            # number of channels specified explicitly
```

...or from a `numpy` array `y` which can be either one-dimensional (single-channel) or two-dimensional (samples-by-channels):

```
s = am.Sound(y, fs=44100)
```

The `Sound` instance's property `s.y` will contain a reference to `y` (or, if `y` was one-dimensional, it will be a two-dimensional samples-by-channels *view* into `y`). You can also initialize from another `Sound` instance in the same way.

Another way to create a `Sound` from scratch is to create an instance and then use its *`GenerateWaveform()`* method:

```
s = am.Sound().GenerateWaveform(freq_hz=440, duration_msec=300)
```

This method is identical to the function of the same name, from the semi-independent *`audiomath.Signal`* submodule. Further useful methods duplicated from that submodule include *`ModulateAmplitude()`* and *`ApplyWindow()`*.

2.4 Define a functionally-generated sound

You can use the `Synth` type (or even a pre-configured `Synth` *instance*) as a function decorator. That will turn the function into a `Synth` instance, which is a duck-typed replacement for a `numpy` array. It can be used to create a `Sound` either directly, or implicitly by passing it to the `Player` constructor:

```
import numpy as np
TWO_PI = 2.0 * np.pi

@am.Synth(fs=22050)
def tone440Hz(fs, sampleIndices, channelIndices):
    timeInSeconds = sampleIndices / fs
    return np.sin(TWO_PI * 440 * timeInSeconds)

s = am.Sound(tone440Hz)  # although it doesn't really support numeric manipulation...

# - or - #

p = am.Player(tone440Hz) # ... so this is the more likely use-case
```

The function should take three arguments: a scalar sampling frequency, an `m x 1` array of sample indices, and a `1 x n` array of channel indices. It should return either an `m x 1` or an `m x n` array of sample values. More options are described in the documentation for the *audiomath.Synth* class.

2.5 Perform simple arithmetic

audiomath takes its name from the ability to do simple arithmetic with `Sound` objects. Sounds can be effortlessly added together (i.e. superimposed, mixed together):

```
s = s1 + s2
```

They can also be multiplied or divided—by scalars, to rescale the sound amplitude:

```
s /= 3
s2 = s * 2
```

—or by sequences of scalars, to rescale individual channels independently:

```
s *= [1, 0.5]
```

—or by other `Sound` objects (to perform amplitude modulation):

```
s = s_carrier * s_envelope
```

...although in that last case you might also want to check out the helper function *audiomath.Signal.ModulateAmplitude()*.

Multiplication by a sequence is a good way to turn a monophonic `Sound` into a stereo or multi-channel `Sound`:

```
s2 = s.Copy().MixDownToMono() * [0, 1]    # mixes everything into the right channel,
                                          # and makes the left channel silent
```

In all cases, the `Sound` object is not overly fussy about matching dimensions: if two sounds of unequal duration are to be added or multiplied together, silence is automatically appended to the shorter data array so that the lengths match. Similarly, if one of the sounds has one channel and the other has more, the monophonic data array is automatically replicated up to the required number of channels. An exception is only raised if both `Sound` objects have more than one channel and the numbers of channels are unequal, or if they have different sampling frequencies.

The in-place versions the operators (`+=`, `-=`, `*=`, `/=`) modify the `Sound` instance (and where possible the `numpy` array `s.y` inside it) in place, whereas the corresponding infix operators (`+`, `-`, `*`, `/`) create a new instance, with a copy of the `numpy` array.

2.6 Extract a segment of a sound

`Sound` objects may be indexed using `slice` subscripts. The numeric start and stop values are interpreted as time in seconds:

```
s2 = s[1:3]    # returns a new `Sound` instance containing a "view" into a
               # two-second segment of `s`, starting 1 second from the beginning

s[-3:] *= 2    # doubles the amplitude of the last three seconds of `s` in-place
```

Start and stop times may also be expressed as strings with the format `MM:SS.FFF` or even `HH:MM:SS.FFF`:

```
s['02:01' : '02:03.4']   # slice from 121s to 123.4s
```

2.7 Extract selected channels from multi-channel sounds

An optional second subscript allows you to refer to specific channels:

```
s2 = s[:, 0]          # returns a new `Sound` instance containing a "view" into just
                      # the first channel of `s`

s2 = s[:, :2]         # returns a new `Sound` instance containing a "view" into just
                      # the first two channels of `s`

s2 = s[:, [0,1]]      # same as above, but returns a copy rather than a view (this is
                  # consistent with the behaviour of the underlying `numpy` array,
                  # when indexed the same way)
```

. . . and manipulate/assign to specific channels:

```
s[:, ::2] = 0         # silence every second channel of `s`, in-place

s[:, -1] = s[:, 0]    # make the last channel of `s` identical to the first
```

Channel indices may also be strings—if so, they are interpreted as 1-based indices rather than 0-based:

```
s2 = s[:, 0]          # \_ equivalent
s2 = s[:, "1"]        # /

s2 = s[:, [1,0]]      # \_ equivalent
s2 = s[:, "21"]       # /
```

You may, of course, slice out a time segment at the same time as extracting channels:

```
s[:0.25, :2] *= 2     # double the amplitude of the first quarter-second of `s`, in
                      # just the first two channels
```

You can also use the *`SplitChannels()`* method to obtain views into different channels or batches of channels.

2.8 Concatenate sounds in time (splicing)

The `%` operator can be used to concatenate `Sound` instances:

```
s = s1 % s2           # creates a new `Sound` instance
s %= s2               # changes instance `s` in-place
```

Numeric scalars stand for the corresponding number of seconds of silence:

```
s2 = 1.23 % s         # create a new instance, with 1.23s of silence prepended
s %= 3.45             # change instance `s` by appending 3.45s of silence
```

Equivalently, you can also use the *`Concatenate()`* method or global `Concatenate()` function. Both of these automatically delve into `list` or `tuple` instances included in their argument lists, which means you can use them with or without `[]` encasing the to-be-concatenated arguments:

```
s = am.Concatenate(s1, 5, s2)       # \_ equivalent (create new instance)
s = am.Concatenate([s1, 5, s2])     # /

s.Concatenate(5, s2)                # \_ equivalent (extend instance `s`)
s.Concatenate([5, s2])              # /
```

In all cases, single-channel data will get automatically replicated up to the required number of channels when you concatenate it with a stereo or multi-channel object. But an exception will be raised if you attempt to concatenate stereo or multi-channel objects that have different numbers of channels.

An exception is also raised if you try to concatenate `Sound` objects that have unequal sampling frequencies. You can use the *`Resample()`* method to equalize them—for example:

```
s  =  s1 % s2.Copy().Resample(s1)
```

Note that, if you want to append or prepend silence, with the goal of ensuring a particular final duration, this is a "padding" operation, which can be performed more easily as follows:

```
s.PadStartTo(2)  # prepend silence to ensure a total length of at least 2 seconds
```

or:

```
s.PadEndTo(3)  # append silence to `s` to ensure a total length of at least 3 seconds

s.PadEndTo(s2) # append silence to `s` to ensure total length is at least that of `s2`
```

2.9 Stack channels (multiplexing)

`Sound` objects can be assembled by "stacking", to create objects with larger numbers of channels. The `&` operator is the easiest way of doing this:

```
s = s1 & s2    # stack two `Sound` instances to create a third instance

s &= s1        # stack the channels of `s1` onto the channels of `s` in-place
```

An exception is raised if you try to stack `Sound` objects that have unequal sampling frequencies. You can use the *`Resample()`* method to equalize them—for example:

```
s  =  s1 & s2.Copy().Resample(s1)
```

Mismatched durations are not a problem—`Sound` data are simply padded with silence at the end to ensure the required length. This means that, by default, content in different channels is aligned in time at the start. To align differently, you can combine your stacking operation with concatenation or padding:

```
s = s1 & (2 % s2)    # stack `s1` with a 2-second-delayed version of `s2`

s &= s1.Copy().PadStartTo(s.duration)  # stack extra channels onto `s`,
                                       # aligned at the end
```

The functional form of the stacking operator is the *`Stack()`* method (for in-place modification of a `Sound` instance) or global `Stack()` function (for creating a new instance). Both of these automatically delve into `list` or `tuple` instances included in their argument lists, so you can use them with or without `[]` encasing the arguments:

```
s = am.Stack(s1, s2, s3)       # \_ equivalent (create new instance)
s = am.Stack([s1, s2, s3])     # /

s.Stack(s1, s2)                # \_ equivalent (modify instance `s`)
s.Stack([s1, s2])              # /
```

2.10 Pitch-shift or time-stretch a sound

audiomath does not itself contain a phase vocoder implementation, but it provides convenience wrappers around the implementation in the optional third-party Python package `librosa` (which you must install separately, for example by saying `python -m pip install librosa` from your shell command prompt).

To use this, you must explicitly `import audiomath.StretchAndShift`. This will automatically import `librosa` and will then give you access to two new `Sound` instance methods:

```
s.TimeStretch(speed=0.5)     # slows down `s`, doubling its duration without
                             # changing its pitch

s.PitchShift(semitones=-12) # pitch-shifts `s` down by 1 octave, without changing
                             # its speed or duration
```

2.11 Preprocess a sound using SoX

The `sox` class (and similarly the `ffmpeg` class) can be used in two ways. The first way is to create an instance—this is an open connection to a running instance of the corresponding `sox` (or `ffmpeg`) command-line binary, good for processing chunks of sound data sequentially and (typically) streaming it out to a file. The second way, which is often simpler, is to use the *class* method *Process()*—this creates a temporary instance, streams to a temporary file, then loads the resulting file contents back into memory. This allows you to transform a `Sound` instance according to the effects provided by SoX, and immediately examine or further manipulate the results—for example:

```
s = am.TestSound('12')
s2 = am.sox.Process(s, effects='loudness -10')
# The `effects` arguments are as they would appear on
# the `sox` command line (see the SoX dox)
```

To take the same example one stage further, the following will generally equalize the perceived loudness (according to ISO 226) across all channels of a Sound `s`:

```
s = am.TestSound().AutoScale()  # eight-channel Sound
s2 = am.Stack(
    am.sox.Process(eachChannel, effects='loudness -10')
    for eachChannel in s.SplitChannels()
).AutoScale() # finally rescale all channels together
              # according to their collective maximum
```

2.12 Manipulate a sound in other miscellaneous ways

The following methods are also useful for tailoring sound objects:

- *AutoScale()* removes DC offsets from each channel, and rescales the sound to standardize its maximum amplitude.
- *Center()* removes DC offsets from each channel.
- *Fade()* fades a sound in at the beginning and/or out at the end.
- *MixDownToMono()* converts the data to mono (single-channel) by averaging across all channels.
- *Resample()* changes the sampling frequency.
- *Reverse()* reverses the sound samples in time.
- *Trim()* removes data below a specified amplitude from the beginning and/or end of a sound.

All methods that modify a `Sound` instance in-place also return a reference to the `Sound` instance itself. This means methods can be chained—for example:

```
s.AutoScale().Trim().Resample(44100)                    # `s` is modified in-place
```

This architecture, in combination with the *Copy()* method, also makes it easy to operate on a copy of a `Sound` instance *without* modifying the original instance in-place, and assign the result to a new variable name at the end of the chain:

```
s2 = s.Copy().AutoScale().Trim().Resample(44100)     # `s` is unchanged
```

2.13 Plot a sound

The third-party package `matplotlib` is used for plotting. Unlike `numpy` (a hard dependency without which *audiomath* can do nothing), `matplotlib` is an optional part of *audiomath*, so it is not installed automatically. If you want to be able to plot, you must install it yourself—for example by typing `python -m pip install matplotlib` at your shell command prompt. Once you have that, plotting is as easy as:

```
s.Plot()
```

Note that, if there are multiple channels, they are vertically separated from each other on the plot so that they can be easily distinguished. If you want a more conventional plot in which they are superimposed (so then the y-axis is more directly interpretable) you can use *audiomath.Signal.PlotSignal* instead:

```
from audiomath.Signal import PlotSignal
s = am.TestSound('12')
PlotSignal(s, hold=False)
```

You can also visualize the amplitude spectrum of the sound:

```
from audiomath.Signal import PlotSpectrum
s = am.TestSound('1')
PlotSpectrum(s, hold=False, dB=True, xlim=[0,1000], ylim=[-120, 0], xlabel='Frequency␣
↪(Hz)', ylabel='Power (dB)')
```

2.14 Play sounds

The programmers' interface to playback and recording is implemented as a generic pure-Python "front end", and the default "back end" is the third-party PortAudio library, binaries for which are included with audiomath for the most

common operating systems. Other back ends may be added in future, with the aim of preserving the programmers' interface unchanged.

The quickest/dirtiest way to hear a preview of a `Sound` instance is to call its *`Play()`* method:

```
s.Play()
```

Note that this is synchronous—the method returns only when the `Sound` ends, or when you press ctrl-C. Note also that it is verbose, as a visible reminder that it is also inefficient: on the console, you will see that a `Player` object is being constructed (which takes a certain amount of time) and then destroyed at the end. For asynchronous, lower-latency playback you will want to create a `Player` instance `p` yourself in advance, and then call `p.Play()` at the time-critical moment. You can construct a `Player` using an existing `Sound` instance:

```
p = am.Player(s)
```

or indeed from anything that would also be a valid input to the `Sound` constructor, such as a filename:

```
p = am.Player('some_file.wav')
```

or a number of seconds of silence:

```
p = am.Player(6)
```

or a `numpy` array. In all cases the current `Sound` instance is then addressable as `p.sound`.

The seconds-of-silence example may seem useless until you know that a `Player` actually contains a `Queue` of `Sound` instances, each of which can be thought of as one "track". When you have multiple tracks, silences then provide one easy way to create gaps between them. Multiple `Sound` instances can be loaded into a `Player` as a `list` or `tuple`:

```
p = am.Player([s1, s2, s3])                # a list of `Sound` instances...
p = am.Player([s, 0.5, 'some_file.wav'])   # or any valid `Sound`-constructor argument
```

...and even from a file `glob` pattern:

```
p = am.Player('my_sounds/*.wav')
```

In all cases, all tracks' sound data are fully represented in memory (all files get loaded and decoded from disk into memory, and all numeric values get fully expanded into arrays of zeros in memory). The attribute `p.queue` is an instance of type `Queue`, which is a container for the tracks' corresponding `Sound` instances. The `Queue` provides a `list`-like interface—for example, you can *`insert()`*, *`append()`*, `del` or *`pop()`* elements:

```
p = am.Player([])      # no tracks
p.queue.append(s)
p.queue.append(0.5)
p.queue.append('some_file.wav')
print(p.queue)    # pretty-prints indices and labels, and marks which track is current
```

To start playing, call `p.Play()` or set `p.playing = True`. To pause, call `p.Pause()` or set `p.playing = False`. During a call to `p.Play()` or `p.Pause()`, or alternatively during the `Player()` constructor call or `p.Set()`, you can use optional keyword arguments to set properties of `p` at the same time. Properties that affect playback behavior are as follows:

- *`loop`* (bool): whether or not to play the current track repeatedly on a (seamless, indefinite) loop;
- *`repeatQueue`* (bool): how to handle attempts to advance beyond the end, or go back earlier than the beginning, of `p.queue` (wrap around, or raise an exception);

- *autoAdvance* (False, True, 'play' or 'pause'): how to behave when the current track p.sound finishes playing (True and 'play' are synonymous);
- *track* (int, str): the integer index of the current track p.sound within the queue p.queue (when assigning this, you can also refer to Sound instances by their label property, e.g.: p.track = 'test12345678');
- *head* (float): current playback position, in seconds, within the current track (negative numbers are counted backwards from the end of the track);
- *speed* (float): a multiplier for playback speed;
- *pan* (float): one way of panning the audio from left (-1.0) to right (+1.0);
- *levels* (list): a list of per-channel amplitude scaling factors, each in the range 0.0 to 1.0 (independent of *pan* and *volume*);
- *volume* (float): over-all amplitude scaling factor (0.0 to 1.0)
- *playing* (bool): reflects and determines the current play/pause status;

So for example, to play a Sound instance s asynchronously on an endless seamless loop, the following are all equivalent:

```
p = am.Player(s);  p.Play(loop=True)

p = am.Player(s, loop=True);  p.Play()

p = am.Player(s);  p.loop = True;  p.Play()

p = am.Player(s, loop=True, playing=True)
```

By contrast, the following options let you play through a whole queue repeatedly on an endless loop (note that this will not be precisely seamless—there is sometimes an unavoidable millisecond or so of silence between tracks):

```
p = am.Player('tracks/*.mp3', loop=False, repeatQueue=True, autoAdvance=True)
p.Play()
```

By default, playback is asynchronous—i.e., p.Play() returns immediately. However if you want synchronous playback, you can say:

```
p.Play(wait=True)     # synchronous playback
```

You can skip to the beginning of the next, or to the beginning of the previous, track in p.queue as follows:

```
p.NextTrack()      #  \_ equivalent
p.track += 1       #  /

p.PreviousTrack()  #  \_ equivalent
p.track -= 1       #  /
```

Most Player properties support dynamic assignment—that means, you can assign a callable function to them, which specifies the property value as a function of time. The following example uses dynamic assignment to the pan property, to pan a sound smoothly and automatically from left to right:

```
p = am.Player(am.TestSound('12').MixDownToMono())
import math
periodInSeconds = 4.0
p.Play(loop=True, pan=lambda t: math.sin(2 * math.pi * t / periodInSeconds))
```

Querying `p.pan` will give you the instantaneous numeric value. The callable itself can be retrieved by `p.GetDynamic('pan')` or by examining the `p.dynamics` property. More information can be obtained with `help(am.Player.dynamics)`.

The *`Play()`* method has low computational overhead: once you have pre-initialized a `Player` instance `p`, you can therefore call `p.Play()` in time-critical parts of your code. If for any reason you need to go one step further, and *construct* new `Player` instances at time-critical moments, you can reduce the initialization overhead by pre-initializing a single `Stream` instance and then passing it as the `stream` argument in all new `Player` initializations:

```
s = am.TestSound('12') # sound pre-loaded into memory
m = Stream()           # connection to audio hardware pre-initialized
                       # (use the `device` keyword here to target a particular device)

p = am.Player(s, stream=m)  # this constructor call is now faster
```

Multiple `Player` instances can (and do, by default) all use the same `Stream` instance if one has already been created. If you want different `Player` instances to target different sound devices simultaneously, you will have to create a `Stream` instance explicitly for each device (see below for instructions on how to *target a particular input or output device*) and assign the streams explicitly when creating the respective `Player` instances.

2.15 Play sounds with more-precise latency, via `PsychPortAudio`

The Psychophysics Toolbox (a.k.a. PsychToolbox) has long been a mainstay of (Matlab-based) stimulus presentation in neuroscience. In 2019 its audio library `PsychPortAudio` was ported to Python as part of the `psychtoolbox` Python package. This is a specially tuned version of the PortAudio library, optimized for low latency and low jitter, with extra functionality such the ability to pre-schedule a sound. *`audiomath`* has a separate "back-end" that can take advantage of this. First, you will need to install the `psychtoolbox` package, for example by typing `python -m pip install psychtoolbox` at your command prompt. Then, in your Python program, you will need to tell audiomath to use the corresponding back-end implementation:

```
import audiomath as am
am.BackEnd.Load('PsychToolboxInterface')
# as opposed to the default 'PortAudioInterface'
```

This sacrifices some functionality—most of the real-time manipulation capabilities, such as dynamic properties and on-the-fly resampling—in return for latency improvements. In January–February 2020 our tests found that the absolute latencies improved a little (macOS, Linux, some Windows devices) or not at all (most of our Windows devices) relative to the best available settings of the default `PortAudioInterface` back-end. However, what *really* improved was the jitter, when taking advantage of the unique "pre-scheduling" functionality `PsychPortAudio` provides. This is implemented via the new `when` property of the `Player` class:

```
am.BackEnd.Load('PsychToolboxInterface')
p = am.Player(am.TestSound('12'))
offsetSeconds = 0.025
t0 = am.Seconds()
p.Play(when=t0 + offsetSeconds)
```

`offsetSeconds` needs to be comfortably larger than the minimal absolute latency `PsychPortAudio` can achieve on your hardware and OS—we found 25ms worked for most of our Windows devices, and 5ms for our Macs. Provided this is the case, we found the time between assignment to `t0` and physical delivery of the sound was accurate to within a millisecond or two (depending on the hardware and OS) and *very* precise: on any given machine, the standard deviation might be of the order of 0.1 milliseconds.

2.16 Record a sound into memory

To record sound data from an input device into memory, the most useful tool is probably the synchronous global `Record()` function:

```
s = am.Record()    # records for up to 60 seconds (less if ctrl-C
                   # is pressed) and returns a `Sound` instance
                   # containing the recorded data.
```

The length of the buffer can be specified by a numeric first argument, or you can record directly into a pre-prepared `Sound`. For example, to record for just 3 seconds:

```
s = am.Record(3, nChannnels=1)                    # \_ equivalent
s = am.Sound(3, nChannnels=1); am.Record(s)       # /
```

Under the hood, the `Record()` function creates an instance of the `Recorder` class. If you want to perform asynchronous recording, you can use this class yourself:

```
r = am.Recorder(10, start=False)

# wait until you are ready to start, then:

r.Record()
while r.recording:
    time.sleep(0.001)  # or use the time however you like...
r.sound.Plot()
```

2.17 Record a sound directly to file

The `filename` argument to `Record()` or `Recorder()` allows you to stream recorded data directly to a file. This requires the command-line utility `ffmpeg` to be separately installed on your system—see the *audiomath.ffmpeg* doc for more details.

To record indefinitely into a file without filling up memory, you can combine this with the `loop=True` option, which lets you record into a circular buffer. The following example records indefinitely into a 3-second circular buffer, streaming the recorded data all the while into the file `blah.mp3`:

```
s2 = am.Record(3, loop=True, filename='blah.mp3')
```

When you press ctrl-C it stops, cleans up, and returns the last 3 seconds as a `Sound` instance.

2.18 Target a particular input or output device

You can have *audiomath* list your devices right from the shell command prompt—for example, on a Mac:

```
$ python -m audiomath --devices

hostApi.name  index     name                  #in  #out  defaultSampleRate
Core Audio    0      <  Built-in Microphone    2    0    44100
Core Audio    1      >  Built-in Output        0    2    44100
Core Audio    2         ZoomAudioDevice        2    2    48000
```

or on Windows:

```
> python -m audiomath --devices

hostApi.name         index     name                                          #in  #out ␣
↪defaultSampleRate
MME                  0         Microsoft Sound Mapper - Input                 2    0   ␣
↪44100
MME                  1       < Microphone (High Definition Aud               2    0   ␣
↪44100
MME                  2         Microphone (High Definition Aud               2    0   ␣
↪44100
MME                  3         Microsoft Sound Mapper - Output               0    2   ␣
↪44100
MME                  4       > Speakers (High Definition Audio               0    2   ␣
↪44100
Windows DirectSound  5       < Primary Sound Capture Driver                  2    0   ␣
↪44100
Windows DirectSound  6         Microphone (High Definition Audio Device)     2    0   ␣
↪44100
Windows DirectSound  7         Microphone (High Definition Audio Device)     2    0   ␣
↪44100
Windows DirectSound  8       > Primary Sound Driver                         0    2   ␣
↪44100
Windows DirectSound  9         Speakers (High Definition Audio Device)       0    2   ␣
↪44100
ASIO                 10      * ASIO4ALL v2                                  2    2   ␣
↪44100
ASIO                 11        Realtek ASIO                                 2    2   ␣
↪44100
Windows WASAPI       12      > Speakers (High Definition Audio Device)      0    2   ␣
↪48000
Windows WASAPI       13        Microphone (High Definition Audio Device)    2    0   ␣
↪44100
Windows WASAPI       14      < Microphone (High Definition Audio Device)    2    0   ␣
↪44100
Windows WDM-KS       15      < Microphone (HD Audio Microphone)             2    0   ␣
↪44100
Windows WDM-KS       16      > Speakers (HD Audio Speaker)                  0    2   ␣
↪44100
Windows WDM-KS       17        Microphone (HD Audio Microphone 2)           2    0   ␣
↪44100
```

The equivalent Python code is:

```
list_of_devices = am.GetDeviceInfo()
print(list_of_devices)      # pretty-prints the list of device records as a table
```

A device record from this list, or even just the integer device index from it, can be passed as the optional `device=` argument to the `Player()`, `Recorder()` and `Stream()` constructors, or to the `Record()` global function.

You can also use `FindDevice()` and `FindDevices()` to narrow down the list according to a partial device name and/or support for a specified number of input and output channels, and to reorder the devices according to your preferred order of host APIs:

```
list_of_devices = am.FindDevices(mode='oo')  # match all devices that provide at
                                             # least two output channels

device = am.FindDevice(mode='oo')    # find the preferred stereo output device (raise
                                     # an exception if no matching device is found)
```

`FindDevice` is called implicitly if you use a string argument as your `device` argument, during `Player`, `Recorder` or `Stream` creation, and it is able to match partial device names, partial host-API names, or a combination of both delimited by `'//'`:

```
p = am.Player(s, device='Speakers') # matches the first device with 'Speakers' in
                                   # its name (host APIs are prioritized according to
                                   # `am.PORTAUDIO.DEFAULT_OUTPUT_API_PREFERENCE_
↪ORDER`)
# - or - #

p = am.Player(s, device='WDM-KS//Speakers')    # Matches host API and device explicitly

# - or - #

p = am.Player(s, device='WDM-KS//')  # default output device of the WDM-KS host API
```

If `p` is the first `Player` instance to be created, it will implicitly create a `Stream` instance, which it will store as `p.stream`. Subsequent newly-constructed `Player` instances will re-use the same `Stream` by default (unless a `Stream` instance is explicitly supplied in the `Player()` constructor) so they will ignore the `device` string. This will continue to be the case until `p.stream` is garbage-collected (i.e. after deletion of `p` and all the other `Player` instances that use the same `Stream`).

2.19 Manipulate the operating-system's overall volume settings

The recommended way to manipulate the system's overall volume is with the *audiomath.SystemVolume.SystemVolumeSetting* context manager:

```
p = am.Player(am.TestSound())

from audiomath.SystemVolume import SystemVolumeSetting
# NB: the sub-module must be imported explicitly

with SystemVolumeSetting(0.1, mute=False):
    p.Play(wait=True)
```

Also, `MAX_VOLUME` is a ready-made `SystemVolumeSetting` instance that can be used to turn the volume up to 1.0 and ensure the output is not muted:

```
from audiomath.SystemVolume import MAX_VOLUME

with MAX_VOLUME:
    p.Play(wait=True)
```

On Windows, this functionality requires the third-party packages `comtypes` and `psutil`, which should have been installed automatically as part of the `python -m pip install audiomath` procedure.

You can also use the lower-level functions *audiomath.SystemVolume.SetVolume()* and *audiomath.SystemVolume.GetVolume()*. However, the context-manager strategy is more highly recommended, as it ensures that the system volume settings are restored to their previous state when the context ends.

2.20 Measure audio latency

For this, you will need the full *audiomath* source-code repository, as described in the *Advanced Installation* section. You will also need to build or adapt the appropriate hardware, described in `python/developer-tests/`

`PlayerLatency-TeensySketch` within the repository. Then you will need to run the script `python/developer-tests/PlayerLatency.py` (run it with `--help` first, for an overview).

CHAPTER 3

API reference

3.1 The `audiomath` package

- *Summary*
- *Classes*
- *Global functions and constants*

3.1.1 Summary

audiomath is a package for reading, manipulating, and writing sound waveforms. It also lets you record and play sounds, with a high degree of real-time control over playback.

To manipulate sound waveforms, plot them, and write them to disk, use the *Sound* class. Usually you would initialize a *Sound* instance from a filename, or from a `numpy` array—in the latter case your array would contain floating-point values in the range [-1, +1], arranged samples-by-channels.

To play sounds, use the *Player* class. You can initialize a *Player* instance from a *Sound* instance, or from anything that can be used to create one (e.g. a filename). You can also supply a sequence of *Sound* instances, or a sequence of filenames or a `glob` pattern that matches multiple files: then you can have the *Player* instance manage multiple "tracks". Each *Player* will only play one sound at a time: to overlap sounds, create multiple *Player* instances.

To record a sound, use the *Record* function, or the *Recorder* class if you want to do it asynchronously.

3.1.2 Classes

- *audiomath.Sound* for manipulating, visualizing and writing sounds.
- *audiomath.Synth* for functionally generating sounds on-the-fly instead of storing them as static arrays.

- *audiomath.Player* for playing sounds.
- *audiomath.Queue* for managing playlists (every *Player* automatically has one of these).
- *audiomath.Fader* and *audiomath.Delay* are examples of "dynamic" callable objects that can be assigned to *Player* properties.
- *audiomath.Recorder* for recording sounds—see also the *Record()* function.
- *audiomath.ffmpeg* for streaming audio data to file via the auxiliary third-party utility ffmpeg, from https://ffmpeg.org
- *audiomath.sox* for transforming audio data via the auxiliary third-party utility sox, from https://sox.sourceforge.net

3.1.3 Global functions and constants

audiomath.**Concatenate**(**args*)
Concatenate sounds in time, to make a new *Sound* instance containing a new array.

You can also concatenate *Sound* instances using the `%` operator, and can modifying an existing *Sound* instance `s` (replacing its array `s.y`) using `%=`:

```
s = s1 % s2
s %= s1
```

Note that, in the above examples, either `s1` or `s2` may also be a numeric scalar: `s %= 0.5` appends half a second of silence to `s`.

audiomath.**MakeFall**(*duration*, *fs=1000*, *hann=False*)
Return a single-channel *Sound* object of the specified `duration` in seconds at the specified sampling frequency `fs` in Hz, containing a "fade-out" envelope (i.e. a ramp from 1 to 0). If `hann` is True, use a raised-cosine profile instead of a linear ramp.

See also: *MakeRise()*, *MakePlateau()*, *MakeHannWindow()*

audiomath.**MakeHannWindow**(*duration*, *fs=1000*, *plateau_duration=0*)
Return a single-channel *Sound* object of the specified `duration` in seconds at the specified sampling frequency `fs` in Hz, containing a Hann or Tukey window—i.e. a raised-cosine "fade-in", followed by an optional plateau, followed by a raised-cosine "fade-out".

See also: *MakeRise()*, *MakePlateau()*, *MakeFall()*

audiomath.**MakePlateau**(*duration*, *fs=1000*, *dc=1.0*)
Return a single-channel *Sound* object of the specified `duration` in seconds at the specified sampling frequency `fs` in Hz, containing a constant `dc` value.

See also: *MakeRise()*, *MakeFall()*, *MakeHannWindow()*

audiomath.**MakeRise**(*duration*, *fs=1000*, *hann=False*)
Return a single-channel *Sound* object of the specified `duration` in seconds at the specified sampling frequency `fs` in Hz, containing a "fade-in" envelope (i.e. a ramp from 0 to 1). If `hann` is True, use a raised-cosine profile instead of a linear ramp.

See also: *MakePlateau()*, *MakeFall()*, *MakeHannWindow()*

audiomath.**QueueTest**()
Construct and return an example *Queue* containing example *Sound* instances. Initialize a *Player* with this to test management of multiple tracks by the *Player* class.

audiomath.**Record**(*space=60*, *prompt=None*, *verbose=None*, *cut=True*, *nChannels=None*, *filename=None*, ***kwargs*)

Record and return a *Sound* synchronously. Slightly easier than creating your own *Recorder* instance and working with that (which is what you would have to do if you want to record asynchronously).

Parameters

- **space** (*Sound, float*) – specifies either a pre-existing *Sound* instance into which to record, or a number of seconds. In the latter case a *Sound* instance is created, pre- allocated with the specified amount of space in seconds.
- **prompt** (*None, bool, str, function*) – if this is left as `None`, a default string is composed. If `prompt` is a string, it is printed to the console and the *Recorder* waits until it has finished (or until ctrl-C is pressed). Alternatively, supply a callable whose non-`None` return value signals that the recording should end.
- **verbose** (*bool*) – Passed through to the constructor of the *Recorder* object that is being used.
- **cut** (*bool*) – If true, return a *Cut()* version of the *Sound* when recording stops—i.e. ensure that the duration of the *Sound* does not exceed the duration of the recording. If false, the entire pre-allocated `space` is returned, with the recording at the beginning.
- **nChannels** (*int*) – Optionally specify the number of channels to record (some APIs have crazy defaults, like ALSA's 32 channels - if you're not using one of those APIs, you probably don't need this).
- **filename** (*str*) – Optionally, use the *ffmpeg* class to stream the recorded content to the specified file. This requires the *ffmpeg* binary to be (separately) installed—see the *ffmpeg* class documentation.
- ****kwargs** – Additional keyword arguments are passed through to the constructor of the *Recorder* object (for example, to specify the `device` that should be used).

Examples:

```
import audiomath as am

s = am.Record()
# this records for up to 60 seconds (less if ctrl-C
# is pressed) and returns a `Sound` instance containing
# the recorded data.

s2 = am.Record(10, loop=True, filename='blah.mp3')
# this records indefinitely into a 10-second circular
# buffer, streaming the recorded data all the while into
# the file `blah.mp3`, and when you press ctrl-C it
# stops, cleans up, and returns the last 10 seconds as
# a `Sound` instance.
```

audiomath.**Stack**(**args*)

Stack multiple Sound objects to make a new multi-channel *Sound* object (appending silence to the end of each argument where necessary to equalize lengths).

You can also stack channels of *Sound* instances using the `&` operator, and can modifying an existing *Sound* instance `s` (replacing its array `s.y`) using `&=`:

```
s = s1 & s2
s &= s1
```

audiomath.**TestSound**(**channels*)

Return an 8-channel *Sound* object, read from a `wav` file that is bundled with *audiomath*, suitable for testing multi-channel sound output devices. Optionally, you can select particular channels using the same argument conventions as the *IsolateChannels()* method. For example, you could choose to return only the first two channels, as follows:

```
s = TestSound('12')
```

audiomath.**ToneTest**(*freq=441*, *amplitude=0.1*, *nChannels=2*, *waveform=<function SineWave>*, ***kwargs*)

This diagnostic test function creates a looped *Player* whose *Sound* is a 1-second pure tone, sampled at the preferred frequency of the *Player* (or the `fs` keyword argument passed through to it, if you want to specify that). Use an integer frequency `freq` to ensure glitch-free looping—any remaining glitches and crackles you hear when you *Play()* are likely the result of buffer underflows in the audio driver. This can happen if the same Python process that is driving the sound is also working too hard on other tasks: higher latencies protect against this.

3.2 The Sound class

class audiomath.**Sound**(*source=None*, *fs=None*, *nChannels=2*, *dtype=None*, *bits=None*, *label=None*)

Bases: `object`

Sound is a class for editing and writing sound files. (If your main aim is simply to play back existing sounds loaded from a file, you probably want to start with a *Player* instead.)

A *Sound* instance `s` is a wrapper around a numpy array `s.y` which contains a floating-point representation of a (possibly multi-channel) sound waveform. Generally the array values are in the range [-1, +1].

The wrapper makes it easy to perform certain common editing and preprocessing operations using Python operators, such as:

Numerical operations:

- The + and − operators can be used to superimpose sounds (even if lengths do not match).
- The `*` and `/` operators can be used to scale amplitudes (usually by a scalar numeric factor, but you can also use a list of scaling factors to scale channels separately, or window a signal by multiplying two objects together).
- The `+=`, `-=`, `*=` and `/=` operators work as you might expect, modifying a *Sound* instance's data array in-place.

Concatenation of sound data in time:

The syntax `s1 % s2` is the same as `Concatenate( s1, s2 )`: it returns a new *Sound* instance containing a new array of samples, in which the samples of `s1` and `s2` are concatenated in time.

Either argument may be a scalar, so `s % 0.4` returns a new object with 400 msec of silence appended, and `0.4 % s` returns a new object with 400 msec of silence pre-pended.

Concatenation can be performed in-place with `s %= arg` or equivalently using the instance method `s.Concatenate( arg1, arg2, ... )`: in either case the instance `s` gets its internal sample array replaced by a new array.

Creating multichannel objects:

The syntax `s1 & s2` is the same as `Stack( s1, s2 )`: it returns a new *Sound* instance containing a new array of samples, comprising the channels of `s1` and the channels of `s2`. Either one may be automatically padded with silence at the end as necessary to ensure that the lengths match.

Stacking may be performed in-place with `s1 &= s2` or equivalently with the instance method `s1.Stack( s2, s3, ... )`: in either case instance `s1` gets its internal sample array replaced by a new array.

Slicing, expressed in units of seconds:

The following syntax returns Sound objects wrapped around slices into the original array:

```
s[:0.5]   #  returns the first half-second of `s`
s[-0.5:]  #  returns the last half-second of `s`

s[:, 0]   # returns the first channel of `s`
s[:, -2:] # returns the last two channels of `s`
s[0.25:0.5, [0,1,3]] # returns a particular time-slice of the chosen␣
↪channels
```

Where possible, the resulting *Sound* instances' arrays are *views* into the original sound data. Therefore, things like `s[2.0:-1.0].AutoScale()` or `s[1.0:2.0] *= 2` will change the specified segments of the *original* sound data in `s`. Note one subtlety, however:

```
# Does each of these examples modify the selected segment of `s` in-place?

    s[0.1:0.2,  :] *= 2             # yes
    q = s[0.1:0.2,  :];  q *= 2     # yes  (`q.y` is a view into `s.y`)

    s[0.1:0.2, ::2] *= 2            # yes
    q = s[0.1:0.2, ::2];  q *= 2    # yes  (`q.y` is a view into `s.y`)

    s[0.1:0.2, 0] *= 2              # yes (creates a copy, but then uses `_
↪_setitem__` on the original)
    q = s[0.1:0.2, 0];  q *= 2      # - NO (creates a copy, then just␣
↪modifies the copy in-place)

    s[0.1:0.2, [1,3]] *= 2          # yes (creates a copy, but then uses `_
↪_setitem__` on the original)
    q = s[0.1:0.2, [1,3]];  q *= 2 # - NO (creates a copy, then just␣
↪modifies the copy in-place)
```

A *Sound* instance may be constructed in any of the following ways:

```
s = Sound( '/path/to/some/sound_file.wav' )
s = Sound( some_other_Sound_instance )    # creates a shallow copy
s = Sound( y, fs )                        # where `y` is a numpy array
s = Sound( duration_in_seconds, fs )      # creates silence
s = Sound( raw_bytes, fs, nChannels=2 )
```

Parameters

- **source** – a filename, another *Sound* instance, a `numpy` array, a scalar numeric value indicating the desired duration of silence in seconds, or a buffer full of raw sound data, as in the examples above.
- **fs** (*float*) – sampling frequency, in Hz. If `source` is a filename or another *Sound* instance, the default value will be inferred from that source. Otherwise the default value is 44100.

- **nChannels** (*int*) – number of channels. If `source` is a filename, another *Sound* instance, or a `numpy` array, the default value will be inferred from that source. Otherwise the default value is 2.
- **dtype** (*str*) – Sound data are always represented internally in floating-point. However, the `dtype` argument specifies the *Sound* instance's `dtype_encoded` property, which dictates the format in which the instance imports or exports raw data by default.
- **bits** (*int*) – This is another way of initializing the instance's `dtype_encoded` property (see `dtype`, above), assuming integer encoding. It should be 8, 16, 24 or 32. If `dtype` is specified, this argument is ignored. Otherwise, the default value is 16.
- **label** (*str*) – An optional string to be assigned to the `label` attribute of the *Sound* instance.

Amplitude (*norm=2*, *threshold=0.0*)
Returns a 1-D `numpy` array containing the estimated amplitude of each channel. With `norm=2`, that's the root-mean-square amplitude, whereas `norm=1` would get you the mean-absolute amplitude.

Note that, depending on the content, the mean may reflect not only how loud the content is *when* it happens, but also how often it happens. For example, loud speech with a lot of pauses might have a lower RMS than continuous quiet speech. This would make it difficult to equalize the volumes of the two speech signals. To work around this, use the `threshold` argument. It acts as a noise-gate: in each channel, the average will only include samples whose absolute value reaches or exceeds `threshold` times that channel's maximum (so the `threshold` value itself is relative, expressed as a proportion of each channel's maximum—this makes the noise-gate invariant to simple rescaling).

ApplyWindow (*func=<function Hann>*, *axis=0*, ***kwargs*)
If `s` is a `numpy.ndarray`, return a windowed copy of the array. If `s` is an *audiomath.Sound* object (for example, if this is being used a method of that class), then its internal array `s.y` will be replaced by a windowed copy.

Windowing means multiplication by the specified window function, along the specified time `axis`.

`func` should take a single positional argument: length in samples. Additional `**kwargs`, if any, are passed through. Suitable examples include `numpy.blackman`, `numpy.kaiser`, and friends.

AutoScale (*max_abs_amp=0.95*)
Remove the median from each channel (see *Center()*) and then rescale the waveform so that its maximum absolute value (across all channels) is `max_abs_amp`. The array `self.y` is modified in place.

Bulk (*encoded=False*)
Return the number of bytes occupied by the sound waveform...

encoded=False: ...currently, in memory.

encoded=True: ...if it were to be encoded according to the currently-specified `dtype_encoded` and written to disk in an uncompressed format (excluding the bytes required to store any format header).

Cat (**args*)
Concatenate the instance, in time, with the specified arguments (which may be other *Sound* instances and/or numpy arrays and/or numeric scalars indicating durations of silent intervals in seconds). Replace the sample array `self.y` with the result. Similar results can be obtained with the `%=` operator or the global function *Concatenate()*.

Center ()
Remove the DC offset from each channel by subtracting the median value.

Concatenate (**args*)
Concatenate the instance, in time, with the specified arguments (which may be other *Sound* instances and/or numpy arrays and/or numeric scalars indicating durations of silent intervals in seconds). Replace

the sample array `self.y` with the result. Similar results can be obtained with the `%=` operator or the global function *`Concatenate()`*.

`Copy`(*empty=False*)
Create a new *`Sound`* instance whose array *`y`* is a deep copy of the original instance's array. With `empty=True` the resulting array will be empty, but will have the same number of channels as the original.

Note that most other preprocessing methods return `self` as their output argument. This makes it easy to choose between modifying an instance in- place and creating a modified copy. For example:

```
s.Reverse()              # reverse `s` in-place
t = s.Copy().Reverse()   # create a reversed copy of `s`
```

`Cut`(*start=None*, *stop=None*, *units='seconds'*)
Shorten the instance's internal sample array by taking only samples from `start` to `stop`. Either endpoint may be `None`. Either may be a positive number of seconds (measured from the start) or a negative number of seconds (measured from the end).

`Detect`(*threshold=0.05*, *center=True*, *p=0*, *eachChannel=False*, *units='seconds'*)
This method finds time(s) at which the absolute signal amplitude exceeds the specified `threshold`.

If `center` is truthy, subtract the median of each channel first.

`p` denotes the location of interest, as a proportion of the duration of the above-threshold data. So, 0.0 means "the first time the signal exceeds the threshold", 1.0 means "the last time the signal exceeds the threshold", and 0.5 means halfway in between those two time points. `p` may be a scalar, in which case the method returns one output argument. Alternatively `p` may be a sequence, in which case a sequence of output arguments is returned, each one corresponding to an element of `p`.

If `eachChannel` is truthy, each output will itself be a sequence (one element per channel). If it is untruthy, each output will be a scalar (the signal is considered to have exceeded the threshold when *any* of its channels exceeds the threshold).

`units` may be `'seconds'`, `'milliseconds'` or `'samples'` and it dictates the units in which the outputs are expressed.

`Duration`()
Returns the duration of the sound in seconds.

`Envelope`(*granularity=0.002*, *fs=None*, *includeDC=False*, *minThickness=0.01*, *bars=False*)
This is both a global function (working on a `numpy` array `s` together with a sampling frequency *`fs`*) and a *`Sound`* method (working on a *`Sound`* instance `s`, in which case no separate *`fs`* argument is required).

It returns `(timebase, lower, upper)`, where `lower` and `upper` are the lower and upper bounds on the signal amplitude. Amplitude os computed in adjacent non-overlapping bins, each bin being of width `granularity` (expressed in seconds).

`lower` and `upper` will be adjusted such that they are always at least `minThickness` apart. Also, if you supply per-channel values as `includeDC`, then the bounds will be adjusted such that the specified values are always included. (If you simply specify `includeDC=True`, then the per-channel median values of `s` will be used in this way.)

The return values are suitable for plotting as follows:

```
s = TestSound('12')
t, lower, upper = s.Envelope()
import matplotlib.pyplot as plt
for iChannel in range(s.nChannels):
        plt.fill_between(t, lower[:, i], upper[:, i])
```

(NB: You do not actually need to plot it by hand in this way, because the *Sound.Plot* method will do it for you: "envelope mode" is used by default for sounds longer than 60 seconds, and this decision can be overridden in either direction by passing `envelope=True` or `envelope=False`.)

By default, `t` is strictly increasing. But if you set `bars=True` then values will be repeated in `t`, `lower` and `upper` such that only horizontal and vertical edges will appear in the plots (Manhattan skyline style).

Fade(*risetime=0*, *falltime=0*, *hann=False*)

If `risetime` is greater than zero, it denotes the duration (in seconds) over which the sound is to be faded-in at the beginning.

If `falltime` is greater than zero, it denotes the duration of the corresponding fade-out at the end.

If `hann` is true, then a raised-cosine function is used for fading instead of a linear ramp.

The array `self.y` is modified in-place.

GenerateWaveform(*freq_hz=1.0*, *phase_rad=None*, *phase_deg=None*, *amplitude=1.0*, *dc=0.0*, *samplingfreq_hz=None*, *duration_msec=None*, *duration_samples=None*, *axis=None*, *waveform=None*, *waveform_domain='auto'*, ***kwargs*)

Create a signal (or multiple signals, if the input arguments are arrays) which is a function of time (time being defined along the specified `axis`).

If this is being used as a method of an *audiomath.Sound* instance, then the `container` argument is automatically set to that instance. Otherwise (if used as a global function), the `container` argument is optional—if supplied, it should be a *audiomath.Sound* object. With a `container`, the `axis` argument is set to 0, and the container object's sampling frequency number of channels and duration (if non-zero) are used as fallback values in case these are not specified elsewhere. The resulting signal is put into `container.y` and a reference to the `container` is returned.

Default phase is 0, but may be changed by either `phase_deg` or `phase_rad` (or both, as long as the values are consistent).

Default duration is 1000 msec, but may be changed by either `duration_samples` or `duration_msec` (or both, as long as the values are consistent).

If `duration_samples` is specified and `samplingfreq_hz` is not, then the sampling frequency is chosen such that the duration is 1 second—so then `freq_hz` can be interpreted as cycles per signal.

The default `waveform` function is `numpy.cos` which means that amplitude, phase and frequency arguments can be taken straight from the kind of dictionary returned by `fft2ap()` for an accurate reconstruction. A `waveform` function is assumed by default to take an input expressed in radians, unless the first argument in its signature is named `cycles`, `samples`, `seconds` or `milliseconds`, in which case the input argument is adjusted accordingly to achieve the named units. (To specify the units explicitly as one of these options, pass one of these words as the `waveform_domain` argument.)

In this module, `SineWave()`, `SquareWave()`, `TriangleWave()` and `SawtoothWave()` are all functions of cycles (i.e. the product of time and frequency), whereas `Click()` is a function of milliseconds. Any of these can be passed as the `waveform` argument.

IsolateChannels(*ind*, **moreIndices*)

Select particular channels, discarding the others. The following are all equivalent, for selecting the first two channels:

```
t = s.IsolateChannels(0, 1)        # ordinary 0-based indexing
t = s.IsolateChannels([0, 1])
t = s.IsolateChannels('1', '2')    # when you use strings, you
t = s.IsolateChannels(['1', '2']) # can index the channels
t = s.IsolateChannels('12')        # using 1-based numbering
```

Equivalently, you can also use slicing notation, selecting channels via the second dimension:

```
t = s[:, [0, 1]]
t = s[:, '12'] # again, strings mean 1-based indexing
```

Left()
Return a new *Sound* instance containing a view of alternate channels, starting at the first.

MakeHannWindow(*plateau_duration=0*)
Return a single-channel *Sound* object of the same duration and sampling frequency as `self`, containing a Hann or Tukey window—i.e. a raised-cosine "fade-in", followed by an optional plateau, followed by a raised- cosine "fade-out".

MixDownToMono()
Average the sound waveforms across all channels, replacing `self.y` with the single-channel result.

ModulateAmplitude(*freq_hz=1.0*, *phase_rad=None*, *phase_deg=None*, *amplitude=0.5*, *dc=0.5*, *samplingfreq_hz=None*, *duration_msec=None*, *duration_samples=None*, *axis=None*, *waveform=None*, ***kwargs*)
If `s` is a `numpy.ndarray`, return a modulated copy of the array. If `s` is an *audiomath.Sound* object (for example, if this is being used a method of that class), then its internal array `s.y` will be replaced by a modulated copy.

Modulation means multiplication by the specified `waveform`, along the specified time `axis`.

Default phase is such that amplitude is 0 at time 0, which corresponds to phase_deg=-90 if `waveform` follows sine phase (remember: by default the modulator is a raised waveform, because `dc=0.5` by default). To change phase, specify either `phase_rad` or `phase_deg`.

Uses *GenerateWaveform()*

NumberOfChannels()
Returns the number of channels.

NumberOfSamples()
Returns the length of the sound in samples.

PadEndTo(*seconds*)
Append silence to the instance's internal array as necessary to ensure that the total duration is at least the specified number of `seconds`.

PadStartTo(*seconds*)
Prepend silence to the instance's internal array as necessary to ensure that the total duration is at least the specified number of `seconds`.

Play(**pargs*, ***kwargs*)
This quick-and-dirty method allows you to play a *Sound*. It creates a *Player* instance in verbose mode, uses it to play the sound, waits for it to finish (or for the user to press ctrl-C), then destroys the *Player* again.

You will get a better user experience, and better performance, if you explicitly create a *Player* instance of your own and work with that.

Arguments are passed through to the *Player.Play()* method.

Plot(*zeroBased=False*, *maxDuration=None*, *envelope='auto'*, *title=True*, *timeShift=0*, *timeScale=1*, *hold=False*, *finish=True*)
Plot the sound waveform for each channel. The third-party Python package `matplotlib` is required to make this work— you may need to install this yourself.

Parameters

- **zeroBased** (*bool*) – This determines whether the y-axis labels show zero-based channel numbers (Python's normal convention) or one-based channel numbers (the convention

followed by almost every other audio software tool). In keeping with audiomath's slicing syntax, zero-based indices are expressed as integers whereas one-based channel indices are expressed as string literals.

- **maxDuration** (*float, None*) – Long sounds can take a prohibitive amount of time and memory to plot. If `maxDuration` is not `None`, this method will plot no more than the first `maxDuration` seconds of the sound. If this means some of the sound has been omitted, a warning is printed to the console.
- **envelope** (*bool, float, 'auto'*) – With `envelope=True`, plot the sound in "envelope mode", which is less veridical when zoomed-in but which uses much less time and memory in the graphics back end. With `envelope=False`, plot the sound waveform as a line. With the default value of `envelope='auto'`, only go into envelope mode when plotting more than 60 seconds' worth of sound. You can also supply a floating-point value, expressed in seconds: this explicitly enforces envelope mode with the specified bin width.
- **title** (*bool, str*) – With `title=True`, the instance's `label` attribute is used as the axes title. With `title=False` or `title=None`, the axes title is left unchanged. If a string is supplied explicitly, then that string is used as the axes title.
- **timeShift** (*float*) – The x-axis (time) begins at this value, expressed in seconds.
- **timeScale** (*float*) – After time-shifting, the time axis is multiplied by this number. For example, you can specify `timeScale=1000` to visualize your sound on a scale of milliseconds instead of seconds.
- **hold** (*bool*) – With `hold=False`, the axes are cleared before plotting. With `hold=True`, the plot is superimposed on top of whatever is already plotted in the current axes.

Read(*source*, *raw_dtype=None*)

Parameters

- **source** – A filename, or a byte string containing raw audio data. With filenames, files are decoded according to their file extension, unless the `raw_dtype` argument is explicitly specified, in which case files are assumed to contain raw data without header, regardless of extension.
- **raw_dtype** (*str*) – If supplied, `source` is interpreted either as raw audio data, or as the name of a file *containing* raw audio data without a header. If `source` is a byte string containing raw audio data, and `raw_dtype` is unspecified, `raw_dtype` will default to `self.dtype_encoded`. Examples might be `float32` or even `float32*2`—the latter explicitly overrides the current value of `self.NumberOfChannels()` and interprets the raw data as 2-channel.

Resample(*newfs*)

Change the instance's sampling frequency. Replace its internal array with a new array, interpolated at the new sampling frequency `newfs` (expressed in Hz).

Reverse()

Reverse the sound in time. The array `self.y` is modified in-place.

Right()

Return a new *Sound* instance containing a view of alternate channels, starting at the second (unless there is only one channel, in which case return that).

SamplesToSeconds(*samples*)

Convert samples to seconds given at the sampling frequency of the instance. `samples` may be a scalar, or a sequence or array. The result is returned in floating-point.

See also: *SecondsToSamples()*

SecondsToSamples (*seconds*, *rounding='round'*)
Convert seconds to samples given the *Sound* instance's sampling frequency. `seconds` may be a scalar, or a sequence or array. The `rounding` approach may be `'floor'`, `'round'`, `'ceil'`, `'none'` or `'int'`. The `'int'` option rounds in the same way as `'floor'` but returns integers—all other options return floating-point numbers.

See also: *SamplesToSeconds()*

SplitChannels (*nChannelsEach=1*)
Return a list of new *Sound* instances, each containing a view into the original data, each limited to `nChannelsEach` consecutive channels.

Stack (**args*)
Stack the instance, across channels, with the specified arguments (which may be other *Sound* instances and/or numpy arrays). Replace the sample array `self.y` with the result. Similar results can be obtained with the `&=` operator or the global function *Stack()*.

Trim (*threshold=0.05*, *tailoff=0.2*, *buildup=0*)
Remove samples from the beginning and end of a sound, according to amplitude.

The new waveform will start `buildup` seconds prior to the first sample on which the absolute amplitude in any channel exceeds `threshold`. It will end `tailoff` seconds after the *last* sample on which `threshold` is exceeded.

copy (*empty=False*)
Create a new *Sound* instance whose array *y* is a deep copy of the original instance's array. With `empty=True` the resulting array will be empty, but will have the same number of channels as the original.

Note that most other preprocessing methods return `self` as their output argument. This makes it easy to choose between modifying an instance in- place and creating a modified copy. For example:

```
s.Reverse()              # reverse `s` in-place
t = s.Copy().Reverse()   # create a reversed copy of `s`
```

dat2str (*data=None*, *dtype=None*)
Converts from a `numpy.array` to a string. `data` defaults to the whole of `s.y`

The string output contains raw bytes which can be written, for example, to an open audio stream.

PitchShift
To use the *PitchShift* method of the *Sound* class, you must first explicitly `import audiomath.StretchAndShift`.

TimeStretch
To use the *TimeStretch* method of the *Sound* class, you must first explicitly `import audiomath.StretchAndShift`.

bits
Bit depth of each sample in each channel, *when encoded* (not necessarily as represented in memory for manipulation and visualization).

bytes
Number of bytes used to represent each sample of each channel, *when encoded* (not necessarily as represented in memory for manipulation and visualization).

duration
Returns the duration of the sound in seconds.

fs
Sampling frequency, in Hz.

nChannels
Returns the number of channels.

nSamples
Returns the length of the sound in samples.

nbits
Bit depth of each sample in each channel, *when encoded* (not necessarily as represented in memory for manipulation and visualization).

nbytes
Number of bytes used to represent each sample of each channel, *when encoded* (not necessarily as represented in memory for manipulation and visualization).

nchan
Returns the number of channels.

nsamp
Returns the length of the sound in samples.

numberOfChannels
Returns the number of channels.

numberOfSamples
Returns the length of the sound in samples.

rms
a 1-D `numpy` array containing root-mean-square amplitude for each channel, i.e. the same as `Amplitude(norm=2, threshold=0)`

y
`numpy` array containing the actual sound sample data.

3.3 The `Synth` class

class audiomath.**Synth**(*func=None*, *fs=44100*, *duration='inf'*, *nChannels=1*, *dtype='float32'*)
Bases: `object`

This class is a duck-typed substitute for the numpy array `s.y` at the heart of a *Sound* instance `s`, but instead of containing static data it allows on-the-fly functional computation of the sound signal.

Example:

```
import numpy as np, audiomath as am
TWO_PI = 2.0 * np.pi
carrierFreq = 440
modFreq = 0.5
@am.Synth
def wobble(fs, samples, channels):
    t = samples / fs
    return np.sin(TWO_PI * carrierFreq * t) * (0.5 - 0.5 * np.cos(TWO_PI *␣
↪modFreq * t))
p = am.Player(wobble)      # could also say am.Sound(wobble)
print(p.sound)
p.Play()
```

You can also use a pre-configured *instance* of *Synth* as the decorator, with the same effect:

```
@am.Synth(fs=22050)
def y(fs, samples, channels):
    t = samples / fs
    return np.sin(2 * np.pi * 440 * t)
p = am.Player(y)      # could also say am.Sound(y)
```

...or the `Use()` method of an instance:

```
y = am.Synth()
@y.Use
def func(fs, samples, channels):   # this way, `func` will be unchanged
    t = samples / fs
    return np.sin(2 * np.pi * 440 * t)
p = am.Player(y)      # could also say am.Sound(y)
```

Parameters

- **func** – A function that takes three arguments: a scalar sampling frequency, an `m x 1` array of sample indices, and a `1 x n` array of channel indices. *`func`* should return an `m x 1` or `m x n` array of sample values.
- **fs** (*float*) – Scalar numeric value specifying the sampling rate.
- **duration** (*float*) – The nominal duration of the sound in seconds. May be infinite.
- **nChannels** (*int*) – Nominal number of channels.
- **dtype** (*str*) – numpy data type for the generated sample data.

static func(*fs*, *samples*, *channels*)

Demo function. Replace this, e.g. by saying `y.func = something_else`, where `y` is your *`Synth`* instance. Or use the `@y.Use` decorator when defining a new function. Or use `@Synth` as the decorator in the first place.

3.4 The `Player` class

class audiomath.**Player**(*sound*, *device=None*, *stream=None*, *bufferLengthMsec=None*, *minLatencyMsec=None*, *fs=None*, *resample=False*, *verbose=None*, ***kwargs*)

Bases: `audiomath.GenericInterface.GenericPlayer`

A *`Player`* provides a persistent connection to the chosen playback hardware, allowing a *`Sound`* instance to be played.

A *`Player`* instance can only play one *`Sound`* at a time, but it can optionally maintain a list of *`Sound`* instances in a *`Queue`* and switch between them. For overlapping sounds, use multiple *`Player`* instances.

A *`Player`* instance can be created directly from a filename, list of filenames or filename glob pattern. The current *`Sound`* instance is available in the *`sound`* property and the complete *`Queue`* is available in the *`queue`* property.

It is better to create a *`Player`* instance than to rely on the quick-and-dirty *`Play()`* methods of the *`Sound`* and *`Queue`* classes—these methods just create a *`Player`* (entailing some computational overhead), wait synchronously for it to finish, and then destroy it again. Creating your own *`Player`* instance provides much greater flexibility and performance.

Parameters

- **sound** (*str, Sound, Queue, None*) – *Sound* instance to play (or sequence of *Sound* instances in a `list` or *Queue*). Alternatively, supply any argument that is also a valid input to the *Sound* or *Queue* constructors (e.g. a filename, list of filenames, or file glob pattern).
- **device** (*int, str, dict, Stream, None*) – Optionally use this argument to specify the device/stream to use for playback—as an integer index, a device name, a full device record from `FindDevice()`, or (fastest of all) an already-open `Stream` instance.
- **stream** (*int, str, dict, Stream, None*) – Synonymous with `device`, for compatibility.
- **bufferLengthMsec** (*float, None, 'auto'*) – Optionally specify a buffer length in milliseconds when creating your first *Player* or first `Stream` (after that, *Player* instances may share an open `Stream` instance so it is possible that only the first call will make any difference). Larger buffer lengths lead to higher playback latencies. `None` means use whatever is currently globally configured in `PORTAUDIO.DEFAULT_BUFFER_LENGTH_MSEC`. `'auto'` or `'pa-default'` means use the default supplied by the PortAudio library.
- **minLatencyMsec** (*float, None, 'auto'*) – Use this setting to override the PortAudio default for "suggested" latency when creating a `Stream`. The values supplied here typically undershoot the empirically measurable latency (in a non-linear fashion) but larger values mean greater robustness (less crackle/stutter susceptibility) at the cost of longer latencies and higher jitter. `None` means use whatever is currently globally configured in `PORTAUDIO.DEFAULT_MIN_LATENCY_MSEC`. `'auto'` means use certain defaults that we have empirically derived to balance these factors. `'pa-default'` means use the defaults supplied by the PortAudio library.
- **fs** (*float, None*) – Optionally specify the sampling frequency, in Hz, when creating your first *Player* or first `Stream` (after that, *Player* instances may share an open `Stream` instance so it is possible that only the first call will make any difference).
- **resample** (*bool*) – Specifies how to handle potential mismatch between the sampling frequency of the sound data `self.sound.fs` and the sampling frequency of the output stream `self.stream.fs`. If true, replace `self.sound` with a copy resampled to the stream's preferred rate. If false, simply adjust playback speed accordingly (at a small, ongoing, computational cost).
- **verbose** (*bool, None*) – Verbosity for debugging. If `None`, inherit from the setting specified by `SetDefaultVerbosity()`, if any.
- ****kwargs** – passed through to *Set()* to initialize properties of the *Player* instance.

GetDynamic(*name*)

For dynamic properties, return the actual callable object that generates property values, rather than the current static value.

Parameters **name** (*str*) – Name of the property

Returns Callable object responsible for generating values for the named property (or `None` if there is no such dynamic).

Example:

```
import audiomath as am, numpy as np
p = am.Player(am.TestSound('12'), loop=True, playing=True)
p.volume = lambda t: 0.5+0.5*np.sin(2.0*np.pi*t*0.25)
v = p.volume                 # this retrieves a float
f = p.GetDynamic('volume')   # this retrieves the lambda
```

NextTrack(*count=1*, *raiseException=True*)

Jump to the beginning of the next track in the *queue* and return the corresponding *Sound* instance.

If this entails going beyond the last track, raise an exception unless `raiseException=False` (in which case go to the beginning of the last track, stop playing, and return `None`).

`Pause`(*position=None*, ***kwargs*)
Stop playing. Optionally change the *`head`* position to the specified absolute `position` in seconds, ready for subsequent playback. Optionally, use keyword arguments to set additional properties (e.g. *`volume`*, *`pan`*, *`levels`*, *`speed`*, *`loop`*, etc) at the same time.

`Play`(*position=None*, *wait=False*, ***kwargs*)
Start playing. Optionally use `position` to specify the exact *`head`* position from which to start playing (in seconds from the beginning of the current *`sound`*). Otherwise playack starts from wherever it last ended up (or at the beginning, if the sound previously ran to the end).

If you call `p.Play()` without an explicit `position` on an instance `p` that is *already* playing, nothing changes—playback simply continues. In this sense it is equivalent to manipulating the *`playing`* property by saying `p.playing = True`.

If `wait` is true, immediately call *`Wait()`* so that playback is synchronous.

Optionally, use further keyword arguments to set additional properties (e.g. *`volume`*, *`pan`*, *`levels`*, *`speed`*, *`loop`*, *`track`*, etc) at the same time.

`PreviousTrack`(*count=1*, *raiseException=True*)
Jump to the beginning of the track prior to the current one in the *`queue`* and return the corresponding *`Sound`* instance.

If this entails going back beyond the beginning, raise an exception unless `raiseException=False` (in which case go to the beginning of the first track, stop playing, and return `None`).

Unlike the "back" button on your music player, this does *not* go to the beginning of the current track before skipping to the previous track. To restart the current track, use `p.Play(0)` or `p.Seek(0)` or `p.head = 0`

`Resample`(*newFs*, *wholeQueue=True*)
Replace `self.sound` with a copy resampled at the new sampling rate `newFs`. Also adjust the Player's internal speed compensation factor, to ensure things sound the same on the existing stream. The most common usage will be to match the sound sample rate to the sample rate of the hardware (so the Player's compensation factor becomes 1.0) to ensure more efficient playback.

`ResetDynamics`(*synchronized=None*)
To restart the clock from which the dynamic properties of a *`Player`* instance `p` are computed, resetting its "time zero" to the current time:

```
p.ResetDynamics()
```

To set the *`synchronizeDynamics`* policy at the same time, pass `True` or `False` as the input argument.

To synchronize `p` to a global "time zero" that multiple *`Player`* instances can share (hence, necessarily, unsynchronized with respect to the playing/pausing of `p` itself):

```
p.ResetDynamics('global')
```

See also: *`dynamics`*, *`synchronizeDynamics`*

`Seek`(*position*, *relative=False*)
Move the playback *`head`* to the specified `position` in seconds. Negative values count back from the end of `self.sound`. If `relative` is true, `position` is interpreted as an offset relative to the current head position. The following are equivalent:

```
p.Seek( 5.0, relative=False )  # more efficient
p.head = 5.0
```

And the following are also equivalent:

```
p.Seek( -2.0,  relative=True ) # more efficient
p.head -= 2.0
```

Set (***kwargs*)

Set the values of multiple attributes or properties in one call. An error will be raised if you try to set the value of a non-existent attribute.

Note that the *Play()* and *Pause()* methods also allow you to set arbitrary properties prior to their manipulation of *playing* status.

Returns `self`

Example:

```
p.Set( head=0, loop=True, volume=0.5 )
```

SetDynamic (*name*, *func*, *order=1*)

Associate a "dynamic" (i.e. a function that can be called repeatedly to set an attribute) with the name of an attribute.

For example:

```
foo.Set( 'bar',  lambda t: t ** 2 )
```

This will set `foo.bar` to `t ** 2` every time the method `foo._RunDynamics( t )` is called (this call will happen automatically from the API user's point of view, in the object's infrastructure implementation).

If the `bar` attribute happens to have been already configured as a "dynamic property" this means it supports "dynamic value assignment", i.e. you can do:

```
foo.bar = lambda t: t ** 2
```

as a syntactic shorthand for *SetDynamic()*—the setter will detect the fact that the value is callable, and divert it to the register of dynamics rather than assigning it directly (so the actual static value you get from querying `foo.bar` will not immediately change).

Parameters

- **name** (*str*) – name of the attribute
- **func** – callable object, or `None` to remove any dynamic that may already be associated with `name`
- **order** (*int, float*) – optional numeric ordering key that allows you to control the serial order in which dynamics are evaluated

Returns `self`

See also: *GetDynamic()*, `GetDynamics()`, `ClearDynamics()`

Stop (*position=None*, ***kwargs*)

Stop playing. Optionally change the *head* position to the specified absolute `position` in seconds, ready for subsequent playback. Optionally, use keyword arguments to set additional properties (e.g. *volume*, *pan*, *levels*, *speed*, *loop*, etc) at the same time.

SwitchToTrack(*position*, *raiseException=True*, *allowNegative=True*)
Immediately jump to the sound at the given position in the *queue* (equivalent to setting the *track* property).

Wait()
Sleep until `self` stops playing, or until the user presses ctrl-C to interrupt.

WaitFor(*condition*, *finalPlayingStatus=None*)
Sleep until `condition` is fulfilled. The `condition` argument may be:

a *Recorder* instance: in which case the method waits until the recorder is no longer recording;

a *Player* instance: in which case the method waits until the player is no longer playing;

a string: in which case the method will print the string as a prompt and wait for the user to enter press ctrl-C on the console;

any callable: in which case the method will repeatedly call `condition()` between 1-ms sleeps until it returns anything other than `None`.

If `finalPlayingStatus` is not `None`, then `self.playing` is set to its value when the wait is over. So for example, you can prompt a user to end ongoing playback as follows:

```
p.WaitFor('Press ctrl-C to stop playing: ', finalPlayingStatus=False)
```

autoAdvance

False: when the current *sound* finishes playing, rewind the *head* to the start of the current sound but do not advance through the *queue*.

True or 'play': when the current *sound* finishes playing, continue to the next *Sound* instance in the *queue*, if any.

'pause': when the current *sound* finishes playing, advance to the next *Sound* instance in the *queue*, if any, but pause playback.

autoAdvance is a dynamic property (see *dynamics*).

See also:

queue, *repeatQueue*, *track*, *NextTrack()*, *PreviousTrack()*

Type This property may take the following values

dynamics
Many *Player* properties, such as *volume*, *speed*, *pan*, *levels* and even *playing*, support "dynamic value assignment". This means you can do something like the following:

```
import audiomath as am
p = am.Player(am.TestSound('12'), loop=True)

p.volume = lambda t: min( 1.0, t / 5.0 )

p.Play()
```

When queried, `p.volume` still appears to be a floating-point number, but this value changes as a function of time `t` in seconds—in this example, the volume fades up from 0.0 to 1.0 over the course of five seconds.

Dynamic functions are called automatically in the streaming callback. You should take care to make them as computationally lightweight and efficient as possible. Your functions may accept 0, 1 or 2 arguments. If they accept 1 argument, it is always time in seconds. If they accept 2 arguments, these will be the *Player* instance first, followed by time in seconds.

Time `t` may be synchronized with, or independent of, the playing/pausing status of the *`Player`* itself, according to the *`synchronizeDynamics`* property. "Time zero" may be reset, or synchronized with other players, using *`ResetDynamics()`*.

Dynamic functions may choose to terminate and remove themselves, by saying `raise StopIteration()` or even `raise StopIteration(final_property_value)`

See also:

`synchronizeDynamics`, *`ResetDynamics()`*, *`GetDynamic()`*, *`SetDynamic()`*

head
Current playback position, in seconds relative to the beginning.

You can change the playback position by assigning to *`head`*, but note that it is fractionally faster/more efficient to do this by calling the *`Seek()`* method.

levels
This is an optional sequence of multipliers applied to each channel in turn. It is independent of (i.e. its effects simply get multiplied by the effects of) other properties that affect intensity, such as *`volume`* and *`pan`*

`levels` is a dynamic property (see *`dynamics`*).

loop
This boolean-valued property determines whether the current sound should wrap around to the beginning when it finishes playing (NB: if you are looping, you will stay on the current *`sound`* and never advance through the *`queue`*).

`loop` is a dynamic property (see *`dynamics`*).

muted
This is a boolean property that, if true, causes the *`Player`* to be silent, independent of the current *`volume`* and/or *`levels`*.

`muted` is a dynamic property (see *`dynamics`*).

norm
For a *`Player`* instance `p`, if you set `p.pan` to a scalar value between -1 and +1, the relative levels of left and right channels are computed such that:

```
left ** p.norm + right ** p.norm = 1
```

The value `p.norm=2` produces a natural-sounding pan but it means that stereo sounds are reduced to 70.71% of their maximum amplitude by default. So instead, the default is to use the infinity-norm, `p.norm='inf'`, which ensures that the larger of the two sides is always 1.0

pan
A value of -1 means left channels only, right channels silent. A value of 0 means left and right channels at equal volume. A value of +1 means right channels only, left channels silent.

The way levels are computed from scalar values between -1 and +1 depends on *`norm`*. Alternatively, you can supply a two- element sequence that explicitly specifies `[left, right]` levels.

Note that these levels are independent from (i.e. they are simply multiplied by) the overall *`volume`* and channel-by- channel *`levels`* settings.

`pan` is a dynamic property (see *`dynamics`*).

playing
This boolean property reflects, and determines, whether the *`Player`* is currently playing or paused.

`playing` is a dynamic property (see *`dynamics`*).

queue
This special sequence of class *Queue* provides a list of *Sound* instances between which the *Player* may switch.

The *Player* instance's *repeatQueue* and *autoAdvance* properties allow you to specify how the queue is used when a given *Sound* finishes playing. The *track* property allows you to jump immediately to a given place in the queue.

NB: direct manipulation of the queue, for example using its own methods `self.queue.Forward()` or `self.queue.Back()`, will *not* affect whichever `self.sound` is currently loaded into the *Player*, but they will affect what happens when that sound has finished playing. If you want to change tracks with immediate effect, you should instead use corresponding *Player* methods `self.NextTrack()` or `self.PreviousTrack()`, or manipulate the property `self.track`.

See also:

autoAdvance, *repeatQueue*, *track*, *NextTrack()*, *PreviousTrack()*

repeatQueue
This boolean property is a shortcut to the sub-property `self.queue.loop`. If true, then moving beyond either end of the *queue* causes a wrap-around to the other end.

repeatQueue is a dynamic property (see *dynamics*).

See also:

autoAdvance, *queue*, *track*, *NextTrack()*, *PreviousTrack()*

sound
A reference to the *Sound* instance that the *Player* is currently playing, or `None`. You can change it on-the-fly. It may also change automatically when the current sound finishes playing, if you have appended additional *Sound* instances the *queue* and have set the *autoAdvance* property.

speed
This floating-point property determines the speed at which the sound should be played. Normal speed is 1.0

speed is a dynamic property (see *dynamics*).

synchronizeDynamics
Many *Player* properties, such as *volume*, *speed*, *pan*, *levels* and even *playing*, can be dynamic, i.e. they can be set to functions of time `t` in seconds.

If `p.synchronizeDynamics` is true, then the dynamic clock pauses whenever playback pauses and resumes only when *playing* becomes true again. Hence, if *playing* itself is dynamic and becomes false, it will not be able to become true again on its own. So this example will only play once:

```
import audiomath as am
p = am.Player( am.TestSound( '12' ) )
p.Play(
        0,
        loop = True,
        synchronizeDynamics = True,
        playing = lambda t: (t / p.sound.duration) % 2 < 1,
)
```

If `p.synchronizeDynamics` is false, then the dynamic clock continues even when playback is paused. Hence, if *playing* is dynamic, the *Player* can stop *and* resume on a schedule. So the same example, with only a change to the *synchronizeDynamics* setting, will play repeatedly, with pauses:

```
import audiomath as am
p = am.Player( am.TestSound( '12' ) )
p.Play(
        0,
        loop = True,
        synchronizeDynamics = False,
        playing = lambda t: (t / p.sound.duration) % 2 < 1,
)
```

A separate but related issue is that of the "time zero" from which dynamic time `t` is measured. By default, time zero for a given *`Player`* instance `p` is the time at which `p` was created. The clock can be reset by using the *`ResetDynamics()`* method. You can also synchronize `p` with a global clock, available to all *`Player`* instances, by saying `p.ResetDynamics('global')`

See also: *`dynamics`*, *`ResetDynamics()`*

track
If there are multiple items in the *`queue`*, this property allows you to switch between them. You may use an integer zero-based index, a *`Sound`* instance, or the string `label` of that instance (provided the instance is already in the *`queue`*)

`track` is a dynamic property (see *`dynamics`*).

See also:

`autoAdvance`, *`queue`*, *`repeatQueue`*, *`NextTrack()`*, *`PreviousTrack()`*

vol
This floating-point property is a multiplier for the amplitude of the sound waveforms. It is independent of (i.e. its effect simply gets multiplied by the effects of) other properties that affect channel-to- channel level differences, such as *`levels`* and *`pan`*

`volume` is a dynamic property (see *`dynamics`*).

volume
This floating-point property is a multiplier for the amplitude of the sound waveforms. It is independent of (i.e. its effect simply gets multiplied by the effects of) other properties that affect channel-to- channel level differences, such as *`levels`* and *`pan`*

`volume` is a dynamic property (see *`dynamics`*).

3.5 The `Queue` class

class audiomath.**Queue**(**sounds*)
Bases: `object`

A `list`-like container for *`Sound`* instances, which keeps track of, and allows manipulation of, which element is considered "current".

You probably do not need to create a *`Queue`* explicitly yourself. When you create a *`Player`* instance `p`, the property `p.queue` will automatically be initialized as a new *`Queue`* containing whatever sounds you specified in the *`Player`* constructor call.

Back(*count=1*)
Move the *`position`* backward by the given count (defaults to 1).

If `self.loop` is False and you try to skip back past the beginning of the queue, a `QueueError` will be raised. Otherwise, it will loop around to the end of the queue.

You probably will not want to call this method directly if this *Queue* is inside a *Player* instance, because it will not automatically update the player's *sound* property. Instead, use the corresponding *Player* method, *PreviousTrack()*

Forward(*count=1*)
Move the *position* ahead by the given count (defaults to 1).

If `self.loop` is False, and you try to skip past the end of the queue, a `QueueError` will be raised Otherwise, it will loop back around to the start of the queue.

You probably will not want to call this method directly if this *Queue* is inside a *Player* instance, because it will not automatically update the player's *sound* property. Instead, use the corresponding *Player* method, *NextTrack()*

MoveSound(*oldIndex*, *newIndex*)
Move sound from `oldIndex` to `newIndex`. Negative indices are acceptable.

Play(**pargs*, ***kwargs*)
This quick-and-dirty method allows you to play a *Queue*. It creates a *Player* instance in verbose mode, configures it with `autoAdvance='play'` by default, then uses it to play a copy of the queue from the beginning, waits for it to finish (or for the user to press ctrl-C), and destroys the *Player* again.

You will get a better user experience, and better performance, if you explicitly create a *Player* instance of your own and work with that.

Arguments are passed through to the *Player.Play()* method.

Restart()
Move *position* back to the first sound.

SetPosition(*position*, *allowNegative=True*)
Move *position* to the specified position.

SwapSounds(*firstIndex*, *secondIndex*)
Swap sound at `firstIndex` with sound at `secondIndex`. Negative indices are acceptable.

add(**sounds*, ***sound_kwargs*)
Add a variable number of sounds to the end of the queue. Since nested lists are not allowed, *add()*, *extend()* and *append()* are identical: any of them will accept a single *Sound* instance, *or* a sequence of *Sound* instances.

Any arguments that are not already *Sound* instances (e.g. filenames) will be passed to the *Sound* constructor to create such instances, along with any additional keyword arguments.

append(**sounds*, ***sound_kwargs*)
Add a variable number of sounds to the end of the queue. Since nested lists are not allowed, *add()*, *extend()* and *append()* are identical: any of them will accept a single *Sound* instance, *or* a sequence of *Sound* instances.

Any arguments that are not already *Sound* instances (e.g. filenames) will be passed to the *Sound* constructor to create such instances, along with any additional keyword arguments.

clear()
Empty the queue.

extend(**sounds*, ***sound_kwargs*)
Add a variable number of sounds to the end of the queue. Since nested lists are not allowed, *add()*, *extend()* and *append()* are identical: any of them will accept a single *Sound* instance, *or* a sequence of *Sound* instances.

Any arguments that are not already *Sound* instances (e.g. filenames) will be passed to the *Sound* constructor to create such instances, along with any additional keyword arguments.

index(*arg*)
Given a *Sound* instance that is in the queue, or its `label`, return its non-negative integer index within the queue. Raise a `ValueError` if no such instance is found in the queue.

insert(*index*, **sounds*, ***sound_kwargs*)
Insert one or more sounds into the queue at given *index*.

Any arguments that are not already *Sound* instances (e.g. filenames) will be passed to the *Sound* constructor to create such instances, along with any additional keyword arguments.

pop(*index*, *default=Unspecified*)
Remove the *Sound* with the given label or index and return it. If it is not found in the list, return `default` if specified (otherwise raise a `KeyError`).

remove(*item_or_index*)
Given either a *Sound* instance, its label, or its index within the list, remove it from the queue. Return `self`.

currentSound
Return the current *Sound* as indexed by *position*

position
Zero-based numeric index of *currentSound* within the list.

When assigning, you may also use a *Sound* instance or its `label`, provided that instance is in the list.

3.6 The `Fader` class

class audiomath.**Fader**(*duration=1.0*, *start=1*, *end=0*, *transform=None*, *pauseWhenDone='auto'*)
Bases: `object`

A callable object that can be assigned to one of the dynamic properties of a *Player* to smoothly transition from a `start` to an `end` value. It is a self- terminating dynamic: when the transition is finished, it will raise a `StopIteration` exception, which will be caught by the *Player*, causing it to replace the dynamic with a final static value.

Example:

```
import time, audiomath as am
p = am.Player(am.TestSound('12'))
fadeIn = am.Fader(3.0, start=0, end=1)
p.Play(loop=1, volume=fadeIn)  # fade in
time.sleep(6.0)                # wait for fade-in, then
                               # stay constant for a few seconds
p.volume = am.Fader(3.0)       # fade out, then stop.
```

Parameters

- **duration** (*float*) – duration of the transition, in seconds
- **start** (*float, numpy.ndarray*) – start value
- **end** (*float, numpy.ndarray*) – end value
- **transform** – an optional function through which to transform `start`, `end` and any value linearly interpolated between them.
- **pauseWhenDone** – if `True`, *Pause()* the *Sound* when the transition is finished. If `False`, do not. A value of `'auto'`, means "pause the sound when the transition is finished if the `end` value is zero".

3.7 The `Delay` class

class audiomath.**Delay**(*seconds*, *initialValue=False*, *finalValue=True*)

Bases: `object`

A callable object that can be assigned to one of the dynamic properties of a *Player*. It returns its `initialValue` until the specified number of `seconds` has elapsed, then raises a `StopIteration` exception containing its `finalValue`. This will be caught by the *Player*, causing it to replace the dynamic with the specified final static value.

The following example causes a *Player* object to start playing automatically after a delay:

```
import audiomath as am
p = am.Player(am.TestSound('12'), synchronizeDynamics=False)

p.playing = am.Delay(5) # will start playing 5 seconds from now
```

3.8 The `Recorder` class

class audiomath.**Recorder**(*seconds*, *device=None*, *stream=None*, *bufferLengthMsec=None*, *minLatencyMsec=None*, *fs=None*, *start=True*, *loop=False*, *verbose=None*, *filename=None*, ***kwargs*)

Bases: `audiomath.GenericInterface.GenericRecorder`

A *Recorder* provides a persistent connection to the chosen recording hardware, allowing sound to be recorded into a *Sound* instance.

You may find it more useful to use the global function *Record*, which synchronously records and returns a *Sound* instance, than to create or interact directly with *Recorder* instances. However, *Recorder* instances are the way to go if you want to record asynchronously, in the background.

Parameters

- **seconds** (*float, Sound*) – number of seconds to pre-allocate for recording, or an already-pre-allocated *Sound* instance into which to record
- **device** (*int, str, dict, Stream*) – specification of the device/stream to use for recording (as an index, name, full device record from `FindDevice()`, or already-open `Stream` instance)
- **stream** (*int, str, dict, Stream*) – synonymous with `device`, for compatibility
- **fs** (*float, None*) – Optionally specify the sampling frequency, in Hz, when creating your first *Recorder* or first `Stream` (after that, *Recorder* instances may share an open `Stream` instance so it is possible that only the first call will make any difference).
- **start** (*bool*) – whether to start recording immediately
- **loop** (*bool*) – whether to record indefinitely, treating `self.sound` as a circular buffer, or simply stop when the capacity of `self.sound` is reached
- **verbose** (*bool, None*) – verbosity for debugging. If `None`, inherit from the setting specified by `SetDefaultVerbosity()`, if any
- ****kwargs** – passed through to *Set()* to initialize properties of the *Recorder* instance.

Cut(*position=None*)

Stop recording, and return the portion of `self.sound` that has been recorded so far, up to the current *head* position (or the explicitly specified `position` in seconds), as a new *Sound* instance.

classmethod MakeRecording(*space=60*, *prompt=None*, *verbose=None*, *cut=True*, *nChannels=None*, *filename=None*, ***kwargs*)

Record and return a *Sound* synchronously. Slightly easier than creating your own *Recorder* instance and working with that (which is what you would have to do if you want to record asynchronously).

Parameters

- **space** (*Sound, float*) – specifies either a pre-existing *Sound* instance into which to record, or a number of seconds. In the latter case a *Sound* instance is created, pre- allocated with the specified amount of space in seconds.
- **prompt** (*None, bool, str, function*) – if this is left as `None`, a default string is composed. If `prompt` is a string, it is printed to the console and the *Recorder* waits until it has finished (or until ctrl-C is pressed). Alternatively, supply a callable whose non-`None` return value signals that the recording should end.
- **verbose** (*bool*) – Passed through to the constructor of the *Recorder* object that is being used.
- **cut** (*bool*) – If true, return a *Cut()* version of the *Sound* when recording stops—i.e. ensure that the duration of the *Sound* does not exceed the duration of the recording. If false, the entire pre-allocated `space` is returned, with the recording at the beginning.
- **nChannels** (*int*) – Optionally specify the number of channels to record (some APIs have crazy defaults, like ALSA's 32 channels - if you're not using one of those APIs, you probably don't need this).
- **filename** (*str*) – Optionally, use the *ffmpeg* class to stream the recorded content to the specified file. This requires the *ffmpeg* binary to be (separately) installed—see the *ffmpeg* class documentation.
- ****kwargs** – Additional keyword arguments are passed through to the constructor of the *Recorder* object (for example, to specify the `device` that should be used).

Examples:

```
import audiomath as am

s = am.Record()
# this records for up to 60 seconds (less if ctrl-C
# is pressed) and returns a `Sound` instance containing
# the recorded data.

s2 = am.Record(10, loop=True, filename='blah.mp3')
# this records indefinitely into a 10-second circular
# buffer, streaming the recorded data all the while into
# the file `blah.mp3`, and when you press ctrl-C it
# stops, cleans up, and returns the last 10 seconds as
# a `Sound` instance.
```

Pause(*position=None*, ***kwargs*)

Stop recording. If `position` is specified, move the recording *head* to that position, expressed in seconds from the beginning.

ReadSamples(*startPositionInSamples*, *nSamples*)

Read the specified number of samples and return them as a `numpy` array. If necessary, wait until the requisite number of samples has been recorded. If `self.loop==False`, then this method may return fewer samples than you ask for if the recording reaches, or has already reached, the end of the space allocated in `self.sound`. Otherwise, the wraparound calculations will be handled automatically and the return value will have the requested number of samples in chronological order.

Record(*position=None*, *wait=False*, ***kwargs*)
Start recording. If `position` is specified, move the recording *head* to that position, expressed in seconds from the beginning, before starting. If `wait` is true, *Wait()* until the end, or until the user presses ctrl+C.

Seek(*position*, *relative=False*)
Move the recording *head* to the specified `position` in seconds. Negative values count back from the end of the available recording space in `self.sound`. If `relative` is true, `position` is interpreted as an offset relative to the current head position. The following are equivalent:

```
r.Seek( 5.0, relative=False )  # more efficient
r.head = 5.0
```

And the following are also equivalent:

```
r.Seek( -2.0,  relative=True ) # more efficient
r.head -= 2.0
```

Set(***kwargs*)
Set the values of multiple attributes or properties in one call. An error will be raised if you try to set the value of a non-existent attribute.

Returns `self`

Example:

```
p.Set( head=0, recording=True )
```

Start(*position=None*, *wait=False*, ***kwargs*)
Start recording. If `position` is specified, move the recording *head* to that position, expressed in seconds from the beginning, before starting. If `wait` is true, *Wait()* until the end, or until the user presses ctrl+C.

Stop(*position=None*, ***kwargs*)
Stop recording. If `position` is specified, move the recording *head* to that position, expressed in seconds from the beginning.

Wait()
Sleep until `self.recording` is no longer true. This will occur when the allocated recording time is exhausted, or if the user presses ctrl+C in the meantime.

WaitFor(*condition*, *finalRecordingStatus=None*)
Sleep until `condition` is fulfilled. The `condition` argument may be:

a *Recorder* instance: in which case the method waits until the recorder is no longer recording;

a *Player* instance: in which case the method waits until the player is no longer playing;

a string: in which case the method will print the string as a prompt and wait for the user to enter press ctrl-C on the console;

any callable: in which case the method will repeatedly call `condition()` between 1-ms sleeps until it returns anything other than `None`.

If `finalRecordingStatus` is not `None`, then `self.recording` is set to its value when the wait is over. So for example, you can prompt a user to end an ongoing recording as follows:

```
r.WaitFor('Press ctrl-C to stop recording: ', finalRecordingStatus=False)
```

head
Current recording position, in seconds relative to the beginning.

loop
A boolean property. If this is `True`, treat `self.sound` as a circular buffer and record into it indefinitely (use the `Cut()` method to extract the recorded data in the correct order). If this is `False`, the recorder will simply stop recording when the capacity of `self.sound` is reached.

3.9 The `ffmpeg` class

class audiomath.**ffmpeg**(*destination*, *source=None*, *format=None*, *nChannels=None*, *fs=None*, *kbps=192*, *transform=None*, *verbose=None*)
Bases: `audiomath.IO.AuxiliaryBinaryInterface`

This class manages the auxiliary command-line binary *ffmpeg* (https://ffmpeg.org).

An instance of this class connects to the standalone *ffmpeg* executable, assuming that that *IsInstalled()*. The instance can then be used to encode audio data to disk in a variety of formats. There are two main applications, illustrated in the following examples. Both of them implicitly use *ffmpeg* instances under the hood:

```
# saving a Sound in a format other than uncompressed .wav:
import audiomath as am
s = am.TestSound( '12' )
s.Write( 'example_sound.ogg' )

# recording direct to disk:
import audiomath as am
s = am.Record(5, loop=True, filename='example_recording.mp3')
```

The latter example uses an *ffmpeg* instance as a callable `hook` of a *Recorder* instance. For more control (asynchronous functionality) you can do it more explicitly as follows:

```
import audiomath as am
h = am.Recorder(5, loop=True, start=False)
h.Record(hook=am.ffmpeg('example_recording.mp3', source=h, verbose=True))

h.Wait() # wait for ctrl-C (replace this line with whatever)
# ...

h.Stop(hook=None) # garbage-collection of the `ffmpeg` instance is one way to `.
↪Close()` it
s = h.Cut()
```

The ffmpeg binary is large, so it is not included in the Python package by default. Installation is up to you. You can install it like any other application, somewhere on the system path. Alternatively, if you want to, you can put it directly inside the *audiomath* package directory, or inside a subdirectory called `aux-bin`—the class method *ffmpeg.Install()* can help you do this. If you install it inside the *audiomath* package, then this has two advantages: (a) audiomath will find it there when it attempts to launch it, and (b) it will be included in the output of `Manifest('pyinstaller')` which you can use to help you freeze your application.

See also:

ffmpeg.Install(), *ffmpeg.IsInstalled()*

Parameters

- **destination** (*str, ffmpeg, sox*) – output filename—be sure to include the file extension so that ffmpeg can infer the desired output format; can alternatively use another instance of

ffmpeg or *sox* as the destination (in that case, `format`, `nChannels` and `fs` will also be intelligently inferred from that instance if they are not otherwise specified);

- **source** (*Sound, Recorder, Player, Stream*) – optional instance that can intelligently specify `format`, `nChannels` and `fs` all in one go;
- **format** (*str*) – format of the raw data—should be either a valid ffmpeg PCM format string (e.g. `'f32le'`) or a construction argument for `numpy.dtype()` (e.g. `'float32'`, `'<f4'`);
- **nChannels** (*int*) – number of channels to expect in each raw data packet;
- **fs** (*int, float*) – sampling frequency, in Hz;
- **kbps** (*int*) – kilobit rate for encoding;
- **transform** (*None, function*) – an optional callable that can receives each data packet, along with the sample number and sampling frequency, and return a transformed version of the data packet to be send to *ffmpeg* in place of the original;
- **verbose** (*bool*) – whether or not to print the standard output and standard error content produced by the binary.

exception NotInstalled(*message=''*)

Bases: `audiomath.IO.NotInstalled`

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

static GrokFormat(*format*)

Convert a format specifier (anything that is accepted as an input to `numpy.dtype()`) into a string that specifies sample format in ffmpeg style (e.g. `'f32le'`).

classmethod Install(*pathToDownloadedExecutable*, *deleteOriginal=False*, *preview=False*)

Auxiliary command-line binaries (like *ffmpeg* or *sox*) can be large relative to the rest of *audiomath*. Therefore, they are not included in the package by default. You should download what you need from the respective project websites (e.g. https://ffmpeg.org or http://sox.sourceforge.net)

If the binary managed by this class is already installed somewhere on the operating-system's search path, *audiomath* will be able to find it, so then you probably do not need this helper function. However, if you have just downloaded, say, a statically-linked build of `ffmpeg.exe`, are looking for somewhere to put it, and want to put it inside the *audiomath* package directory itself, then this function will do it for you (it will actually put it inside an automatically-created sub-directory called `aux-bin`). *audiomath* will find it there, and it will also be included in the output of `Manifest('pyinstaller')` which helps if you want to use `pyinstaller` to freeze your application.

NB: if the utility consists of multiple files (e.g. `sox.exe` and its many associated `dll` files on Windows) then `pathToDownloadedExecutable` should be a path to the *directory* that contains them all.

classmethod IsInstalled()

Return `True` or `False` depending on whether the auxiliary binary executable managed by this class (*ffmpeg* or *sox*) is accessible and executable, either inside the *audiomath* package (see *Install()*) or on the operating system's search path.

See also: *Install()*

classmethod Process(*snd*, *destination=None*, ***kwargs*)

Whereas instances of `AuxiliaryBinaryInterface` subclasses are good for processing chunks of sound data sequentially, if you simply want to process a whole *Sound* instance in one go, it is easier to use this class method. Under the hood, it will create a temporary instance, using any specified `**kwargs`.

You can direct its output to the file specified by `destination`. Alternatively you can leave `destination=None` and thereby receive the output (actually written to and read back from a temporary uncompressed `wav` file) as a new *Sound* instance returned by this function.

Example, using the *sox* subclass:

```
import audiomath as am
s = am.TestSound('12')
s2 = am.sox.Process(s, effects='loudness -10')

# or, to take that example one stage further,
# the following will generally equalize the
# perceived loudness (according to ISO 226)
# across all channels of a Sound `s`:

s = am.TestSound().AutoScale()
s2 = am.Stack(
    am.sox.Process(eachChannel, effects='loudness -10')
    for eachChannel in s.SplitChannels()
).AutoScale() # finally rescale all channels together
              # according to their collective maximum
```

3.10 The sox class

class audiomath.**sox**(*destination*, *source=None*, *format=None*, *nChannels=None*, *fs=None*, *transform=None*, *effects=''*, *verbose=None*)

Bases: `audiomath.IO.AuxiliaryBinaryInterface`

This class manages the auxiliary command-line binary *sox* (https://sox.sourceforge.net).

An instance of this class connects to the standalone *sox* executable, assuming that that *IsInstalled()*. The instance can then be used to transform audio data in a wide variety of ways, and encode it to disk in a variety of formats (although typically fewer formats than *ffmpeg*).

You may find the class method *sox.Process()* more useful than an instance.

You can also chain processing from one *sox* instance to another, or between a *sox* instance and a *ffmpeg* instance, by passing the other instance as this one's `destination` instead of specifying a filename.

Here is an example of chaining. Let's say we want to use a sox effect and save the result as in mp3 format, but sox does not support the mp3 format. So instead we write out, on a pipe, to an *ffmpeg* process, which can save to mp3:

```
import audiomath as am
s = am.TestSound('12')
am.sox.Process(
    s,
    destination=am.ffmpeg( 'output.mp3', source=s ),
    effects='loudness -10',
)
```

The sox binary is not included in the Python package by default. Installation is up to you. You can install it like any other application, somewhere on the system path. Alternatively, if you want to, you can put it directly inside the *audiomath* package directory, or inside a subdirectory called `aux-bin`—the class method *sox.Install()* can help you do this. If you install it inside the *audiomath* package, then this has two advantages: (a) audiomath will find it there when it attempts to launch it, and (b) it will be included in the output of `Manifest('pyinstaller')` which you can use to help you freeze your application.

Parameters

- **destination** (*str, ffmpeg, sox*) – output filename—be sure to include the file extension so that ffmpeg can infer the desired output format; can alternatively use another instance of *ffmpeg* or *sox* as the destination (in that case, `format`, `nChannels` and `fs` will also be intelligently inferred from that instance if they are not otherwise specified);
- **source** (*Sound, Recorder, Player, Stream*) – optional instance that can intelligently specify `format`, `nChannels` and `fs` all in one go;
- **format** (*str*) – format of the raw data—should be either a valid ffmpeg PCM format string (e.g. `'f32le'`) or a construction argument for `numpy.dtype()` (e.g. `'float32'`, `'<f4'`);
- **nChannels** (*int*) – number of channels to expect in each raw data packet;
- **fs** (*int, float*) – sampling frequency, in Hz;
- **effects** (*str, list, tuple*) – a string, or sequence of strings, specifying the effect names and effect options to be passed to sox on the command line;
- **transform** (*None, function*) – an optional callable that can receive each data packet, along with the sample number and sampling frequency, and return a transformed version of the data packet to be send to *sox* in place of the original;
- **verbose** (*bool*) – whether or not to print the standard output and standard error content produced by the binary.

exception NotInstalled(*message=''*)

Bases: `audiomath.IO.NotInstalled`

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

static GrokFormat(*format*)

Convert a format specifier (anything that is accepted as an input to `numpy.dtype()`) into a string that specifies sample format in ffmpeg style (e.g. `'f32le'`).

classmethod Install(*pathToDownloadedExecutable*, *deleteOriginal=False*, *preview=False*)

Auxiliary command-line binaries (like *ffmpeg* or *sox*) can be large relative to the rest of *audiomath*. Therefore, they are not included in the package by default. You should download what you need from the respective project websites (e.g. https://ffmpeg.org or http://sox.sourceforge.net)

If the binary managed by this class is already installed somewhere on the operating-system's search path, *audiomath* will be able to find it, so then you probably do not need this helper function. However, if you have just downloaded, say, a statically-linked build of `ffmpeg.exe`, are looking for somewhere to put it, and want to put it inside the *audiomath* package directory itself, then this function will do it for you (it will actually put it inside an automatically-created sub-directory called `aux-bin`). *audiomath* will find it there, and it will also be included in the output of `Manifest('pyinstaller')` which helps if you want to use `pyinstaller` to freeze your application.

NB: if the utility consists of multiple files (e.g. `sox.exe` and its many associated `dll` files on Windows) then `pathToDownloadedExecutable` should be a path to the *directory* that contains them all.

classmethod IsInstalled()

Return `True` or `False` depending on whether the auxiliary binary executable managed by this class (*ffmpeg* or *sox*) is accessible and executable, either inside the *audiomath* package (see *Install()*) or on the operating system's search path.

See also: *Install()*

classmethod Process(*snd*, *destination=None*, ***kwargs*)

Whereas instances of `AuxiliaryBinaryInterface` subclasses are good for processing chunks of sound data sequentially, if you simply want to process a whole *Sound* instance in one go, it is easier to use this class method. Under the hood, it will create a temporary instance, using any specified `**kwargs`.

You can direct its output to the file specified by `destination`. Alternatively you can leave `destination=None` and thereby receive the output (actually written to and read back from a temporary uncompressed `wav` file) as a new *Sound* instance returned by this function.

Example, using the *sox* subclass:

```
import audiomath as am
s = am.TestSound('12')
s2 = am.sox.Process(s, effects='loudness -10')

# or, to take that example one stage further,
# the following will generally equalize the
# perceived loudness (according to ISO 226)
# across all channels of a Sound `s`:

s = am.TestSound().AutoScale()
s2 = am.Stack(
    am.sox.Process(eachChannel, effects='loudness -10')
    for eachChannel in s.SplitChannels()
).AutoScale() # finally rescale all channels together
              # according to their collective maximum
```

3.11 The audiomath.PortAudioInterface sub-module

This submodule contains PortAudio-specific implementations of high- level *Player* and *Recorder* classes. They inherit their API-user- facing functionality, as well as their code for manipulating raw `Sound` data, from `GenericInterface.GenericPlayer` and `GenericInterface.GenericRecorder`. The PortAudio-specific subclasses act as intermediaries between the generic stuff and the objects and functions in the `_wrap_portaudio` submodule.

This submodule also contains various global functions for listing and selecting the available sound devices and their host APIs.

The `PortAudioInterface` is the default playing/recording back-end for *audiomath*. Therefore, all the symbols exported by this submodule are also available at the top level `audiomath.*` namespace (at least until another back-end is written, and loaded with `audiomath.BackEnd.Load()`).

`audiomath.PortAudioInterface.`**SetDefaultVerbosity**(*value*)

Useful for debugging when the PortAudio library is initialized or terminated, when streams are opened, started, stopped or closed, and when other objects such as players or recorders are initialized or garbage-collected. Objects may be individually marked as `verbose=True` or `verbose=False`, but by default they inherit their verbosity from the setting you specify here.

`audiomath.PortAudioInterface.`**LowLatencyMode**(*turnOn*, *preferASIO=False*)

Call this function before you create your first *Stream* or *Player* instance.

On Windows, `LowLatencyMode(True)` will manipulate `PORTAUDIO.DEFAULT_OUTPUT_API_PREFERENCE_ORDER` to cause the WDM-KS host API (or the ASIO host API, if `preferASIO` is `True`) to be preferred over DirectSound when constructing output streams. This will allow considerably lower output latencies, but will have the side-effect of taking exclusive control of the sound driver, meaning that sounds from other applications will no longer be audible (this may or may not be desirable, depending on your application).

`LowLatencyMode(True)` also sets `PORTAUDIO.DEFAULT_BUFFER_LENGTH_MSEC = None`, and `PORTAUDIO.DEFAULT_MIN_LATENCY_MSEC = 4`, which will have the effect of lowering latencies but at the cost of making sounds vulnerable to stuttering if Python performs any significant work during playback.

`LowLatencyMode(False)` reverts to factory settings (DirectSound is reinstated as the preferred host API on Windows, and `PORTAUDIO.DEFAULT_MIN_LATENCY_MSEC` is set to `'auto'` (which means 60 for DirectSound and at least 10 elsewhere) for future *Stream* instances.

audiomath.PortAudioInterface.**GetHostApiInfo**()
Returns a list of records corresponding to PortAudio's `Pa_GetHostApiInfo()` output for every available host API. You can `print()` the result, or otherwise convert it to `str()`, to see a pretty tabulated summary.

audiomath.PortAudioInterface.**GetDeviceInfo**()
Returns a list of records corresponding to PortAudio's `Pa_GetDeviceInfo()` output for every available combination of host API and device. You can `print()` the result, or otherwise convert it to `str()`, to see a pretty tabulated summary.

audiomath.PortAudioInterface.**FindDevice**(*id=None*, *mode=None*, *apiPreferences=None*, *_candidates=None*)
Returns the first device matched by *FindDevices()* (plural) according to the specified criteria. Raises an exception if there are no matches.

audiomath.PortAudioInterface.**FindDevices**(*id=None*, *mode=None*, *apiPreferences=None*, *_candidates=None*)
Calls *GetDeviceInfo()*, filtering and reordering the outputs according to the specified criteria. You can `print()` the result, or otherwise convert it to `str()`, to see a pretty tabulated summary.

You can use *FindDevice()* (singular) to return just the top-ranking result (and to assert that at least one result can be found).

Parameters

- **id** (*int, dict, str, None*) –
  - `None`: matches all devices;
  - `int`: matches only the device whose `index` field matches `id`;
  - `dict` (including the objects returned by this function, which are `dict` subclassses): matches only the device whose `index` field matches `id['index']`
  - `str`: matches any device with the specified word or phrase in its `name` field (you can also match `hostApi.name` and `name` simultaneously if you delimit them with `'//'`—for example, `'WDM-KS//Speakers'`);
- **mode** (*str, tuple, None*) – May be either a two-element tuple such as `mode=(minInputChannels,minOutputChannels)`, or a string containing a number of `'o'` and/or `'i'` characters. In either case, devices are only matched if they provide at least the specified number of input and output channels. For example, `FindDevices(mode='oo')` matches all devices that provide two or more output channels.
- **apiPreferences** (*str, None*) – If this is left at `None`, it defaults to the current value of `PORTAUDIO.DEFAULT_INPUT_API_PREFERENCE_ORDER` (if the `mode` argument requests any inputs) or `PORTAUDIO.DEFAULT_OUTPUT_API_PREFERENCE_ORDER` otherwise. The string `'*'` matches all host APIs. Host APIs may be comma-delimited. For example, `apiPreferences='DirectSound,*'` means "give first priority to devices hosted by the DirectSound API, and then, failing that, match all other APIs". If `'*'` is not included in the specification, this argument may limit the number of records returned. In any case it will affect the ordering of the returned records.

audiomath.PortAudioInterface.**Tabulate**(*records*, **fields*)

Returns a pretty-printed table of the specified `fields` (i.e. dictionary entries or attributes) of a list of records. This is called automatically, with certain default combinations of field names, when you `print()` the results of *GetHostApiInfo()*, *GetDeviceInfo()* or *FindDevices()*.

class audiomath.PortAudioInterface.**Stream**(*device=None*, *mode=None*, *apiPreferences=None*, *outputCallbacks=None*, *sampleRate=None*, *sampleFormat=None*, *verbose=None*, *bufferLengthMsec=None*, *minLatencyMsec=None*)

A persistent connection to an audio driver. Multiple output callbacks (player objects) and/or multiple input callbacks (recorder objects) may share the same *Stream*, or under some conditions it may be possible for each callback (or object) to have its own *Stream*.

Typically, objects that play or record will automatically create a *Stream* or add themselves to an existing *Stream*, so you will not usually need to construct a *Stream* instance yourself. The exception is when you need to ensure that the construction of such objects is itself fast: then, creating a *Stream* in advance, and keeping a reference to it to use in initializing the other objects, will speed things up.

The `device`, `mode` and `apiPreferences` arguments are the same as `id`, `mode` and `apiPreferences` in *FindDevice()*.

audiomath.PortAudioInterface.**Seconds**()

A platform-dependent high-precision clock. Origin (time zero) is arbitrary, but fixed within a given Python session.

audiomath.PortAudioInterface.**GetOutputDevice**()

Return the default output device, if any has been set.

See also: *SetOutputDevice()*

audiomath.PortAudioInterface.**SetOutputDevice**(*device*)

Set the specified `device` as the default device to be used for *Player* objects. The `device` argument may be the device index, one or more words from the device name, or the full device record from *GetDeviceInfo()* or *FindDevices()*.

Note that, by default, *Player* instances share a single *Stream* instance—this means that the preference you specify here (and the preference you specify in the *Player* constructor) may not be honored if there are already other *Player* instances in existence (and hence an already-running *Stream*). In short: it's best to use this function *before* you construct your first *Player* or *Stream* instance.

See also: *GetOutputDevice()*

class audiomath.PortAudioInterface.**Player**(*sound*, *device=None*, *stream=None*, *bufferLengthMsec=None*, *minLatencyMsec=None*, *fs=None*, *resample=False*, *verbose=None*, ***kwargs*)

A *Player* provides a persistent connection to the chosen playback hardware, allowing a `Sound` instance to be played.

A *Player* instance can only play one `Sound` at a time, but it can optionally maintain a list of `Sound` instances in a `Queue` and switch between them. For overlapping sounds, use multiple *Player* instances.

A *Player* instance can be created directly from a filename, list of filenames or filename glob pattern. The current `Sound` instance is available in the *sound* property and the complete `Queue` is available in the *queue* property.

It is better to create a *Player* instance than to rely on the quick-and-dirty *Play()* methods of the `Sound` and `Queue` classes—these methods just create a *Player* (entailing some computational overhead), wait synchronously for it to finish, and then destroy it again. Creating your own *Player* instance provides much greater flexibility and performance.

Parameters

- **sound** (*str, Sound, Queue, None*) – `Sound` instance to play (or sequence of `Sound` instances in a `list` or `Queue`). Alternatively, supply any argument that is also a valid input to the `Sound` or `Queue` constructors (e.g. a filename, list of filenames, or file glob pattern).
- **device** (*int, str, dict, Stream, None*) – Optionally use this argument to specify the device/stream to use for playback—as an integer index, a device name, a full device record from *FindDevice()*, or (fastest of all) an already-open *Stream* instance.
- **stream** (*int, str, dict, Stream, None*) – Synonymous with `device`, for compatibility.
- **bufferLengthMsec** (*float, None, 'auto'*) – Optionally specify a buffer length in milliseconds when creating your first *Player* or first *Stream* (after that, *Player* instances may share an open *Stream* instance so it is possible that only the first call will make any difference). Larger buffer lengths lead to higher playback latencies. `None` means use whatever is currently globally configured in `PORTAUDIO.DEFAULT_BUFFER_LENGTH_MSEC`. `'auto'` or `'pa-default'` means use the default supplied by the PortAudio library.
- **minLatencyMsec** (*float, None, 'auto'*) – Use this setting to override the PortAudio default for "suggested" latency when creating a *Stream*. The values supplied here typically undershoot the empirically measurable latency (in a non-linear fashion) but larger values mean greater robustness (less crackle/stutter susceptibility) at the cost of longer latencies and higher jitter. `None` means use whatever is currently globally configured in `PORTAUDIO.DEFAULT_MIN_LATENCY_MSEC`. `'auto'` means use certain defaults that we have empirically derived to balance these factors. `'pa-default'` means use the defaults supplied by the PortAudio library.
- **fs** (*float, None*) – Optionally specify the sampling frequency, in Hz, when creating your first *Player* or first *Stream* (after that, *Player* instances may share an open *Stream* instance so it is possible that only the first call will make any difference).
- **resample** (*bool*) – Specifies how to handle potential mismatch between the sampling frequency of the sound data `self.sound.fs` and the sampling frequency of the output stream `self.stream.fs`. If true, replace `self.sound` with a copy resampled to the stream's preferred rate. If false, simply adjust playback speed accordingly (at a small, ongoing, computational cost).
- **verbose** (*bool, None*) – Verbosity for debugging. If `None`, inherit from the setting specified by *SetDefaultVerbosity()*, if any.
- ****kwargs** – passed through to *Set()* to initialize properties of the *Player* instance.

`audiomath.PortAudioInterface.`**`GetInputDevice`**`()`

Return the default input device, if any has been set.

See also: *SetInputDevice()*

`audiomath.PortAudioInterface.`**`SetInputDevice`**`(`*`device`*`)`

Set the specified `device` as the default device to be used for *Recorder* objects. The `device` argument may be the device index, one or more words from the device name, or the full device record from *GetDeviceInfo()* or *FindDevices()*.

See also: *GetInputDevice()*

`class` `audiomath.PortAudioInterface.`**`Recorder`**`(`*`seconds`*`, `*`device=None`*`, `*`stream=None`*`, `*`bufferLengthMsec=None`*`, `*`minLatencyMsec=None`*`, `*`fs=None`*`, `*`start=True`*`, `*`loop=False`*`, `*`verbose=None`*`, `*`filename=None`*`, `*`**kwargs`*`)`

A *Recorder* provides a persistent connection to the chosen recording hardware, allowing sound to be recorded into a `Sound` instance.

You may find it more useful to use the global function `Record`, which synchronously records and returns a `Sound` instance, than to create or interact directly with *`Recorder`* instances. However, *`Recorder`* instances are the way to go if you want to record asynchronously, in the background.

Parameters

- **seconds** (*float, Sound*) – number of seconds to pre-allocate for recording, or an already-pre-allocated `Sound` instance into which to record
- **device** (*int, str, dict, Stream*) – specification of the device/stream to use for recording (as an index, name, full device record from *`FindDevice()`*, or already-open *`Stream`* instance)
- **stream** (*int, str, dict, Stream*) – synonymous with `device`, for compatibility
- **fs** (*float, None*) – Optionally specify the sampling frequency, in Hz, when creating your first *`Recorder`* or first *`Stream`* (after that, *`Recorder`* instances may share an open *`Stream`* instance so it is possible that only the first call will make any difference).
- **start** (*bool*) – whether to start recording immediately
- **loop** (*bool*) – whether to record indefinitely, treating `self.sound` as a circular buffer, or simply stop when the capacity of `self.sound` is reached
- **verbose** (*bool, None*) – verbosity for debugging. If `None`, inherit from the setting specified by *`SetDefaultVerbosity()`*, if any
- ****kwargs** – passed through to *`Set()`* to initialize properties of the *`Recorder`* instance.

3.12 The `audiomath.PsychToolboxInterface` sub-module

This sub-module provides an alternative back-end for high- precision playback, based on the PsychToolbox project's reworking of the PortAudio binaries. It requires the third-party `psychtoolbox` module to be installed (you must do this yourself, for example by running `python -m pip install psychtoolbox`, because `psychtoolbox` is not a hard/automatically-installed dependency of *`audiomath`*.

This alternative back-end can be enabled by calling `audiomath.BackEnd.Load('PsychToolboxInterface')`. This removes the default `audiomath.PortAudioInterface.*` symbols from the top-level `audiomath.*` namespace and replaces them with `audiomath.PsychToolboxInterface.*` symbols, most of which have names and prototypes that are familiar from the default back-end (*`Player`*, *`Stream`*, *`FindDevices`*, etc...)

Extra features:

- The `latencyClass` input parameter to *`Player`* and *`Stream`* constructors. This is only respected for the first such construction call, since the same *`Stream`* will be shared by all *`Player`* instances). The supported settings are all attributes of the *`LATENCY_CLASS`* enumerator. The higher the number, the more aggressively (and less cooperatively) the host API will drive the sound device, and the lower the latency is likely to get.
- The ability to pre-schedule sounds with the `when` argument, e.g. `now = Seconds(); p.Play(when=now + 0.5)`

Limitations, relative the the default `PortAudioInterface`:

- *`Player`* objects cannot `Seek(t)` to any position except the beginning `t=0` (likewise `Play(t)`, `Pause(t)` or `Set(head=t)` will only work if `t` is 0)
- no on-the-fly resampling or *`speed`* control
- no dynamic properties
- no on-the-fly sound synthesis (`Synth` data get frozen and truncated to 10 seconds when passed into a *`Player`*).

- no *Recorder* implementation

Notes

- on Windows, PsychToolbox uses the WASAPI host API by default. For best performance, and especially if you want to use `latencyClass < 2` (such that other processes can access the sound device at the same time) then you should ensure your sounds are sampled at 48000 Hz, not the usual 44100 Hz.
- some host APIs may be unable to be driven with certain latency classes (for example, on Windows most of the soundcard drivers we have tried cannot use WDM-KS with a latency class higher than 1)

`audiomath.PsychToolboxInterface.`**`GetHostApiInfo`**`()`
Returns a list of records corresponding to PortAudio's `Pa_GetHostApiInfo()` output for every available host API. You can `print()` the result, or otherwise convert it to `str()`, to see a pretty tabulated summary.

`audiomath.PsychToolboxInterface.`**`GetDeviceInfo`**`()`
Returns a list of records corresponding to PortAudio's `Pa_GetDeviceInfo()` output for every available combination of host API and device. You can `print()` the result, or otherwise convert it to `str()`, to see a pretty tabulated summary.

`audiomath.PsychToolboxInterface.`**`FindDevices`**`(`*id=None*, *mode=None*, *apiPreferences=None*, *_candidates=None*`)`
Calls *GetDeviceInfo()*, filtering and reordering the outputs according to the specified criteria. You can `print()` the result, or otherwise convert it to `str()`, to see a pretty tabulated summary.

You can use *FindDevice()* (singular) to return just the top-ranking result (and to assert that at least one result can be found).

Parameters

- **id** (*int, dict, str, None*) –
  - `None`: matches all devices;
  - `int`: matches only the device whose `index` field matches `id`;
  - `dict` (including the objects returned by this function, which are `dict` subclassses): matches only the device whose `index` field matches `id['index']`
  - `str`: matches any device with the specified word or phrase in its `name` field (you can also match `hostApi.name` and `name` simultaneously if you delimit them with `'//'`—for example, `'WDM-KS//Speakers'`);
- **mode** (*str, tuple, None*) – May be either a two-element tuple such as `mode=(minInputChannels,minOutputChannels)`, or a string containing a number of `'o'` and/or `'i'` characters. In either case, devices are only matched if they provide at least the specified number of input and output channels. For example, `FindDevices(mode='oo')` matches all devices that provide two or more output channels.
- **apiPreferences** (*str, None*) – If this is left at `None`, it defaults to the current value of `PORTAUDIO.DEFAULT_INPUT_API_PREFERENCE_ORDER` (if the `mode` argument requests any inputs) or `PORTAUDIO.DEFAULT_OUTPUT_API_PREFERENCE_ORDER` otherwise. The string `'*'` matches all host APIs. Host APIs may be comma-delimited. For example, `apiPreferences='DirectSound,*'` means "give first priority to devices hosted by the DirectSound API, and then, failing that, match all other APIs". If `'*'` is not included in the specification, this argument may limit the number of records returned. In any case it will affect the ordering of the returned records.

audiomath.PsychToolboxInterface.**FindDevice**(*id=None*, *mode=None*, *apiPreferences=None*, *_candidates=None*)

Returns the first device matched by *FindDevices()* (plural) according to the specified criteria. Raises an exception if there are no matches.

class audiomath.PsychToolboxInterface.**LATENCY_CLASS**

This class is an enum container: a namespace of values that can be passed as the `latencyClass` constructor argument when creating a `PsychToolboxInterface.Player` instance.

RELAXED (0): indicates that latency is not a priority—latencies are then unpredictable and may be in the hundreds of milliseconds.

SHARED (1): applies the most aggressive latency settings that still allow other processes to play sound at the same time.

EXCLUSIVE (2): takes exclusive control of the sound card.

AGGRESSIVE (3): takes exclusive control of the sound card, with even more aggressive low-latency settings than EXCLUSIVE mode.

CRITICAL (4): is the same as AGGRESSIVE mode, but raises an exception if the settings cannot be achieved.

class audiomath.PsychToolboxInterface.**Stream**(*device=()*, *mode=None*, *sampleRate=None*, *latencyClass='DEFAULT'*, *verbose=False*, *bufferLengthMsec=None*, *minLatencyMsec=None*)

A persistent connection to an audio driver. Multiple output callbacks (player objects) and/or multiple input callbacks (recorder objects) may share the same *Stream*, or under some conditions it may be possible for each callback (or object) to have its own *Stream*.

Typically, objects that play or record will automatically create a *Stream* or add themselves to an existing *Stream*, so you will not usually need to construct a *Stream* instance yourself. The exception is when you need to ensure that the construction of such objects is itself fast: then, creating a *Stream* in advance, and keeping a reference to it to use in initializing the other objects, will speed things up.

The `device`, `mode` and `apiPreferences` arguments are the same as `id`, `mode` and `apiPreferences` in *FindDevice()*.

The `latencyClass` argument is unique to the `PsychToolboxInterface` back-end. It specifies the PsychPortAudio latency mode (use one of the names or integer values defined in the enumerator *LATENCY_CLASS*).

class audiomath.PsychToolboxInterface.**Player**(*sound*, *device=None*, *stream=None*, *latencyClass='DEFAULT'*, *bufferLengthMsec=None*, *minLatencyMsec=None*, *fs=None*, *resample=True*, *verbose=None*, ***kwargs*)

A *Player* provides a persistent connection to the chosen playback hardware, allowing a `Sound` instance to be played.

A *Player* instance can only play one `Sound` at a time, but it can optionally maintain a list of `Sound` instances in a `Queue` and switch between them. For overlapping sounds, use multiple *Player* instances.

A *Player* instance can be created directly from a filename, list of filenames or filename glob pattern. The current `Sound` instance is available in the *sound* property and the complete `Queue` is available in the *queue* property.

It is better to create a *Player* instance than to rely on the quick-and-dirty *Play()* methods of the `Sound` and `Queue` classes—these methods just create a *Player* (entailing some computational overhead), wait synchronously for it to finish, and then destroy it again. Creating your own *Player* instance provides much greater flexibility and performance.

Parameters

- **sound** (*str, Sound, Queue, None*) – `Sound` instance to play (or sequence of `Sound` instances in a `list` or `Queue`). Alternatively, supply any argument that is also a valid input to the `Sound` or `Queue` constructors (e.g. a filename, list of filenames, or file glob pattern).
- **device** (*int, str, dict, Stream, None*) – Optionally use this argument to specify the device/stream to use for playback—as an integer index, a device name, a full device record from *`FindDevice()`*, or (fastest of all) an already-open *`Stream`* instance.
- **stream** (*int, str, dict, Stream, None*) – Synonymous with `device`, for compatibility.
- **latencyClass** (*int, str*) – specify the PsychPortAudio latency mode (use one of the names or integer values defined in the enumerator *`LATENCY_CLASS`*)
- **bufferLengthMsec** (*float, None*) – Optionally specify a buffer length in milliseconds when creating your first *`Player`* or first *`Stream`*. (Future *`Player`* instances will all use the same setting.)
- **minLatencyMsec** (*float, None, 'auto'*) – Optionally specify the value to use as "suggested latency" when creating your first *`Player`* or first *`Stream`*. (Future *`Player`* instances will all use the same setting.)
- **fs** (*float, None*) – Optionally specify the sampling frequency, in Hz, when creating your first *`Player`* or first *`Stream`* (after that, *`Player`* instances may share an open *`Stream`* instance so it is possible that only the first call will make any difference).
- **resample** (*bool*) – This should be left as `True` because there is no on-the-fly resampling capability in the `PsychToolboxInterface` back-end, The parameter is included purely for compatibility with code written for the default `PortAudioInterface` back-end.
- **verbose** (*bool, None*) – Verbosity for debugging. If `None`, inherit from the setting specified by `SetDefaultVerbosity()`, if any.
- ****kwargs** – passed through to *`Set()`* to initialize properties of the *`Player`* instance.

`levels`

This is an optional sequence of multipliers applied to each channel in turn. It is independent of (i.e. its effects simply get multiplied by the effects of) other properties that affect intensity, such as *`volume`* and *`pan`*

`levels` is a dynamic property (see *`dynamics`*).

`muted`

This is a boolean property that, if true, causes the *`Player`* to be silent, independent of the current *`volume`* and/or *`levels`*.

`muted` is a dynamic property (see *`dynamics`*).

`norm`

For a *`Player`* instance `p`, if you set `p.pan` to a scalar value between -1 and +1, the relative levels of left and right channels are computed such that:

```
left ** p.norm + right ** p.norm = 1
```

The value `p.norm=2` produces a natural-sounding pan but it means that stereo sounds are reduced to 70.71% of their maximum amplitude by default. So instead, the default is to use the infinity-norm, `p.norm='inf'`, which ensures that the larger of the two sides is always 1.0

`pan`

A value of -1 means left channels only, right channels silent. A value of 0 means left and right channels at equal volume. A value of +1 means right channels only, left channels silent.

The way levels are computed from scalar values between -1 and +1 depends on *norm*. Alternatively, you can supply a two- element sequence that explicitly specifies `[left, right]` levels.

Note that these levels are independent from (i.e. they are simply multiplied by) the overall *volume* and channel-by- channel *levels* settings.

pan is a dynamic property (see *dynamics*).

sound
A reference to the `Sound` instance that the *Player* is currently playing, or `None`. You can change it on-the-fly. It may also change automatically when the current sound finishes playing, if you have appended additional `Sound` instances the *queue* and have set the *autoAdvance* property.

vol
This floating-point property is a multiplier for the amplitude of the sound waveforms. It is independent of (i.e. its effect simply gets multiplied by the effects of) other properties that affect channel-to- channel level differences, such as *levels* and *pan*

volume is a dynamic property (see *dynamics*).

volume
This floating-point property is a multiplier for the amplitude of the sound waveforms. It is independent of (i.e. its effect simply gets multiplied by the effects of) other properties that affect channel-to- channel level differences, such as *levels* and *pan*

volume is a dynamic property (see *dynamics*).

class audiomath.PsychToolboxInterface.**Recorder**(**p*, ***k*)
TODO: no *Recorder* implementation yet in the *audiomath.PsychToolboxInterface* back-end

classmethod MakeRecording(**p*, ***k*)
TODO: no `Record()` implementation yet in this back-end

audiomath.PsychToolboxInterface.**Seconds**()
An alias for PsychToolbox's clock, `psychtoolbox.GetSecs()`

3.13 The audiomath.Signal sub-module

This semi-independent submodule provides various basic signal-processing tools that can be used with or without the *audiomath.Sound* container.

audiomath.Signal.**UnwrapDiff**(*x*, *base=6.283185307179586*, *axis=None*, *startval=None*, *dtype=None*)
Assume `X` is a wrapped version of an underlying value `Y` we're interested in. For example, it's a 16-bit value that wraps around at 65536, or it's an angle which wraps back to 0 at 2*pi.

`base` is the value (65536 or 2*pi in the above examples) such that `X = Y % base`. The default value of `base` is 2*pi.

Let `dY` be the numeric diff of `Y` in dimension `axis`, computed from `X` by unwrapping in order to avoid jumps larger than `base/2`. Thus, with `base=65536`, a jump from 65535 to 1 is considered as a step of +2. With `base=360`, a jump from 10 to 350 is considered as a step of -20.

`Y` is then reconstructed based on `dY` and `startval` (which defaults to the actual initial value(s) of `X`).

Return value is `(dY,Y)`.

audiomath.Signal.**msec2samples**(*msec*, *samplingfreq_hz*)
Converts milliseconds to the nearest integer number of samples given the specified sampling frequency.

audiomath.Signal.**samples2msec**(*samples*, *samplingfreq_hz*)
Converts samples to milliseconds given the specified sampling frequency.

audiomath.Signal.**ApplyWindow**(*s*, *func=<function Hann>*, *axis=0*, ***kwargs*)
If `s` is a `numpy.ndarray`, return a windowed copy of the array. If `s` is an *audiomath.Sound* object (for example, if this is being used a method of that class), then its internal array `s.y` will be replaced by a windowed copy.

Windowing means multiplication by the specified window function, along the specified time `axis`.

`func` should take a single positional argument: length in samples. Additional `**kwargs`, if any, are passed through. Suitable examples include `numpy.blackman`, `numpy.kaiser`, and friends.

audiomath.Signal.**ModulateAmplitude**(*s*, *freq_hz=1.0*, *phase_rad=None*, *phase_deg=None*, *amplitude=0.5*, *dc=0.5*, *samplingfreq_hz=None*, *duration_msec=None*, *duration_samples=None*, *axis=None*, *waveform=None*, ***kwargs*)
If `s` is a `numpy.ndarray`, return a modulated copy of the array. If `s` is an *audiomath.Sound* object (for example, if this is being used a method of that class), then its internal array `s.y` will be replaced by a modulated copy.

Modulation means multiplication by the specified `waveform`, along the specified time `axis`.

Default phase is such that amplitude is 0 at time 0, which corresponds to phase_deg=-90 if `waveform` follows sine phase (remember: by default the modulator is a raised waveform, because `dc=0.5` by default). To change phase, specify either `phase_rad` or `phase_deg`.

Uses *GenerateWaveform()*

audiomath.Signal.**GenerateWaveform**(*container=None*, *freq_hz=1.0*, *phase_rad=None*, *phase_deg=None*, *amplitude=1.0*, *dc=0.0*, *samplingfreq_hz=None*, *duration_msec=None*, *duration_samples=None*, *axis=None*, *waveform=None*, *waveform_domain='auto'*, ***kwargs*)
Create a signal (or multiple signals, if the input arguments are arrays) which is a function of time (time being defined along the specified `axis`).

If this is being used as a method of an *audiomath.Sound* instance, then the `container` argument is automatically set to that instance. Otherwise (if used as a global function), the `container` argument is optional—if supplied, it should be a *audiomath.Sound* object. With a `container`, the `axis` argument is set to 0, and the container object's sampling frequency number of channels and duration (if non-zero) are used as fallback values in case these are not specified elsewhere. The resulting signal is put into `container.y` and a reference to the `container` is returned.

Default phase is 0, but may be changed by either `phase_deg` or `phase_rad` (or both, as long as the values are consistent).

Default duration is 1000 msec, but may be changed by either `duration_samples` or `duration_msec` (or both, as long as the values are consistent).

If `duration_samples` is specified and `samplingfreq_hz` is not, then the sampling frequency is chosen such that the duration is 1 second—so then `freq_hz` can be interpreted as cycles per signal.

The default `waveform` function is `numpy.cos` which means that amplitude, phase and frequency arguments can be taken straight from the kind of dictionary returned by *fft2ap()* for an accurate reconstruction. A `waveform` function is assumed by default to take an input expressed in radians, unless the first argument in its signature is named `cycles`, `samples`, `seconds` or `milliseconds`, in which case the input argument is adjusted accordingly to achieve the named units. (To specify the units explicitly as one of these options, pass one of these words as the `waveform_domain` argument.)

In this module, *SineWave()*, *SquareWave()*, *TriangleWave()* and *SawtoothWave()* are all functions of cycles (i.e. the product of time and frequency), whereas *Click()* is a function of milliseconds. Any of these can be passed as the `waveform` argument.

`audiomath.Signal.`**`Click`**(*milliseconds*, *positivePulseWidth=0.5*, *negativePulseWidth='symmetric'*)
A signal function that can be passed as the `waveform` argument to *GenerateWaveform*, to generate periodic clicks. Each click is a positive pulse optionally followed by a negative pulse. If `negativePulseWidth` is `None` or the string `'symmetric'`, then the width of the negative pulse is made the same as `positivePulseWidth`. Otherwise, input arguments are all expressed in milliseconds.

`audiomath.Signal.`**`SineWave`**(*cycles*, *phase_deg=0*, *maxharm=None*, *rescale=False*)
A sine wave, but with the input expressed in cycles (0 to 1) instead of radians (0 to 2*pi). Otherwise no different from `numpy.sin`, except that the function signature has `maxharm` and `rescale` arguments (which are ignored) for compatibility with *SquareWave()*, *TriangleWave()* and *SawtoothWave()*.

This function can be passed as the `waveform` argument to *GenerateWaveform*.

`audiomath.Signal.`**`SquareWave`**(*cycles*, *phase_deg=0*, *maxharm=None*, *rescale=False*, *duty=0.5*, *ramp=0*, *tol=1e-08*)
A square wave with its peaks and troughs in sine phase. If `maxharm` is an integer, then an anti-aliased approximation to the square wave (containing no components of higher frequency than `maxharm` times the fundamental) is returned instead. In this case, the `rescale` flag can be set to ensure that the sampled waveform does not exceed +/- 1.0

This function can be passed as the `waveform` argument to *GenerateWaveform*.

`audiomath.Signal.`**`TriangleWave`**(*cycles*, *phase_deg=0*, *maxharm=None*, *rescale=False*)
A triangle wave with its peaks and troughs in sine phase. If `maxharm` is an integer, then an anti-aliased approximation to the triangle wave (containing no components of higher frequency than `maxharm` times the fundamental) is returned instead. The `rescale` flag, included for compatibility with *SawtoothWave()* and *SquareWave()*, has no effect.

This function can be passed as the `waveform` argument to *GenerateWaveform*.

`audiomath.Signal.`**`SawtoothWave`**(*cycles*, *phase_deg=0*, *maxharm=None*, *rescale=False*)
A sawtooth wave with its polarity and zero-crossings in sine phase. If `maxharm` is an integer, then an anti-aliased approximation to the sawtooth wave (containing no components of higher frequency than `maxharm` times the fundamental) is returned instead. In this case, the `rescale` flag can be set to ensure that the waveform does not exceed +/- 1.0

This function can be passed as the `waveform` argument to *GenerateWaveform*.

`audiomath.Signal.`**`Noise`**(*cycles*, *distribution='uniform'*, ***kwargs*)
This function can be passed as the `waveform` argument to *GenerateWaveform*. The `distribution` argument should be a function, or the name of a function within `numpy.random`, that takes a `size` keyword argument dictating the shape of its output array. Additional `**kwargs` are passed through to the function.

If the function is `numpy.random.uniform`, then the `low` argument is set to -1.0 by default instead of the usual 0.0.

If the function is `numpy.random.normal`, then the `scale` argument is set to 0.2 by default instead of the usual 1.0.

`audiomath.Signal.`**`fftfreqs`**(*nSamples*, *samplingfreq_hz=1.0*)
Return a 1-D numpy.array of length `nSamples` containing the positive and negative frequency values corresponding to the elements of an `nSamples`-point FFT. If `samplingfreq_hz` is not supplied, 1.0 is assumed, so the result has 0.5 as its Nyquist frequency.

`audiomath.Signal.`**`fft2ap`**(*X*, *samplingfreq_hz=2.0*, *axis=0*)
Given discrete Fourier transform(s) `X` (with frequency along the specified `axis`), return a dict containing a

properly scaled amplitude spectrum, a phase spectrum in degrees and in radians, and a frequency axis (coping with all the fiddly edge conditions).

The inverse of `d = fft2ap(X)` is `X = ap2fft(**d)`

audiomath.Signal.**ap2fft**(*amplitude*, *phase_rad=None*, *phase_deg=None*, *samplingfreq_hz=2.0*, *axis=0*, *freq_hz=None*, *fullfreq_hz=None*, *nSamples=None*)
Keyword arguments match the fields of the `dict` output by *fft2ap()*.

The inverse of `d = fft2ap(X)` is `X = ap2fft(**d)`

audiomath.Signal.**Reconstruct**(*ap*, ***kwargs*)
Check the accuracy of *fft2ap()* and *GenerateWaveform()* by reconstructing a signal as the sum of cosine waves with amplitudes and phases specified in dict `ap`, which is of the form output by *fft2ap()*.

audiomath.Signal.**Toy**(*nSamples=11*, *nCycles=None*, *amplitude=(1.0, 0.1)*, *phase_deg=0*)
Toy sinusoidal signals for testing *fft2ap()* and *ap2fft()*. Check both odd and even `nSamples`.

audiomath.Signal.**Shoulder**(*x*, *s*, *complement=False*)
Return a (possibly asymmetric) Tukey window function of `x`. `s` may have 1, 2, 3 or 4 elements:

1. raised cosine between `x=s[0]-0.5` and `x=s[0]+0.5`
2. raised cosine between `x=s[0]` and `x=s[1]`
3. raised cosine rise from `s[0]` to `s[1]`, and fall from `s[1]` to `s[2]`
4. raised cosine rise from `s[0]` to `s[1]`, plateau from `s[1]` to `s[2]`, and fall from `s[2]` to `s[3]`

audiomath.Signal.**Hilbert**(*x*, *N=None*, *axis=0*, *band=()*, *samplingfreq_hz=None*, *return_dict=False*)
Compute the analytic signal, just like `scipy.signal.hilbert` but with the differences that (a) the computation can be performed along any axis and (b) a limited band of the signal may be considered. The `band` argument can be a two-, three- or four-element tuple suitable for passing to *Shoulder()*, specifying the edges of the passband (expressed in Hz if `samplingfreq_hz` is explicitly supplied, or relative to Nyquist if not).

If `return_dict` is True, do not return just the complex analytic signal but rather a dict containing its amplitude, phase, and unwrapped phase difference.

audiomath.Signal.**Spectrum**(*x*, *samplingfreq_hz=None*, *axis=None*)
Runs `fft` on a signal followed by *fft2ap*, to return spectral information in "human-readable" format with the annoying corner cases handled.

The inverse operation is *InverseSpectrum()*.

audiomath.Signal.**InverseSpectrum**(*ap*, *real=True*)
Inverse of *Spectrum()*. Takes the `dict` output of *Spectrum()* (or of the underlying *fft2ap()*) and runs *ap2fft()* followed by `ifft()` on it to return a signal. If `real` is true, which it is by default, discard the imaginary part of the result.

audiomath.Signal.**ReweightSpectrum**(*s*, *func*, **pargs*, ***kwargs*)
Brutally filter the sound `s` by transforming into the Fourier domain, weighting the amplitude spectrum, and transforming back. The filtering is non-causal and (TODO) will have wrap-around artifacts whereby the two ends of the signal may bleed into each other, so use with caution—it is most suitable for noise or for periodic signals. No group delay is introduced.

Args:

s (numpy.ndarray or audiomath.Sound): The input sound data. If `s` is an array, a new array will be returned. If it is a `Sound` instance, `s` will get changed in-place (the array `s.y` will be replaced by a new array) and returned.

func (callable): A function that takes a numeric array of frequencies, in Hz, as its first (and only required) argument, and outputs a corresponding array of weights.

samplingfreq_hz (float): Sampling frequencym in Hz. Not needed if `s` is a `Sound` instance.

axis (int): Axis along which time runs. Not needed if `s` is a `Sound` instance.

Additional `*pargs` and `**kwargs` are passed straight through to `func()`.

`audiomath.Signal.`**`Timebase`**(*x*, *samplingfreq_hz=None*, *axis=None*)
Return a discrete time axis in seconds, against which signal `x` can be plotted. `x` may be an *audiomath.Sound* instance, or you may supply `samplingfreq_hz` explicitly.

`audiomath.Signal.`**`PlotSignal`**(*x*, *samplingfreq_hz=None*, *axis=None*, ***kwargs*)
Plot `x` (an array with time running along the specified `axis`, or an *audiomath.Sound* instance) against the appropriate *Timebase()*.

`audiomath.Signal.`**`PlotSpectrum`**(*x*, *samplingfreq_hz=None*, *axis=None*, *dB=False*, *baseValue=None*, ***kwargs*)
Input argument `x` may be a `numpy.array` or *audiomath.Sound*, in which case *Spectrum()* will be called on it. Or it may be the `dict` output of a previous *Spectrum()* or *fft2ap()* call.

3.14 The `audiomath.SystemVolume` sub-module

Utilities for manipulating the operating-system master volume and/or mixer.

- On Windows, this requires the third-party packages `comtypes` and `psutil` (via our fork of Andre Miras' `pycaw`, which is included).
- On macOS, it uses Applescript.
- On Linux, there is a highly-experimental implementation based on `pacmd` and `pactl` (so these PulseAudio command-line utilities must be installed).

Because of its dependencies, this submodule is not imported by default—you must explicitly `import audiomath.SystemVolume`.

The recommended way to use this module is via the *SystemVolumeSetting* context-manager object:

```
import audiomath
from audiomath.SystemVolume import SystemVolumeSetting
p = audiomath.Player(audiomath.TestSound())
with SystemVolumeSetting(0.1, mute=False):
    p.Play(wait=True)
```

`audiomath.SystemVolume.`**`GetVolume`**(*dB=False*, *session='master'*)
Query volume level and mute settings.

If you want to query volume settings, change them, and later change them back, `SystemVolumeSettings` is a more usable option.

The Linux implementation uses Pulse Audio utility `pacmd`. It is likely to break whenever the text format of `pacmd list-sinks` changes in future versions.

`audiomath.SystemVolume.`**`SetVolume`**(*level=None*, *dB=False*, *mute=None*, *session='master'*)
Sets volume level and/or mute status.

Consider using the context-manager class *SystemVolumeSetting* instead. For a description of the input arguments, see *SystemVolumeSetting*.

The Linux implementation uses Pulse Audio utilities `pacmd` and `pactl`. It is likely to break whenever the text format of `pacmd list-sinks` changes in future versions.

class audiomath.SystemVolume.**SystemVolumeSetting**(*level=None*, *dB=False*, *mute=None*, *session='master'*, *verbose=False*)

This class is a context-manager. It allows you to change the system volume temporarily, such that it automatically changes back to its previous setting when you are done:

```
import audiomath
from audiomath.SystemVolume import SystemVolumeSetting
p = audiomath.Player( audiomath.TestSound() )
with SystemVolumeSetting(0.1, mute=False):
    p.Play()
    audiomath.WaitFor(p) # otherwise we'll exit the `with`
                         # clause before anything has been
                         # streamed and won't hear the effect
```

For your convenience, `MAX_VOLUME` is a ready-instantiated *`SystemVolumeSetting`* made with `level=1.0` and `mute=False`:

```
from audiomath.SystemVolume import MAX_VOLUME
with MAX_VOLUME:
    p.Play(wait=True)
```

`SystemVolumeSetting` instances can be combined together with the `&` operator. This is only really useful on Windows where session-by-session control is possible:

```
MUTE_FIREFOX = SystemVolumeSetting(mute=True, session='Mozilla Firefox')
with MAX_VOLUME & MUTE_FIREFOX:
    p.Play(wait=True)
```

Parameters

- **level** (*float, None*) – volume setting, from 0.0 to 1.0 (assuming `dB=False`) or from negative infinity to 0.0 (with `dB=True`). Use `None` if you want to leave the level untouched and manipulate only the `mute` setting.
- **dB** (*bool*) – By default, `level` values are interpreted and expressed as relative power values from 0 to 1. However, with `dB=True` you can choose to have them interpreted and expressed in decibels, i.e. `10*log10(power)`, with 0.0 as the maximum.
- **mute** (*bool, None*) – Whether or not to mute the output. Use `None` to leave the mute setting untouched and manipulate only the `level`.
- **session** (*str, int*) – (Windows only): control one individual "session" within the Windows mixer. A string will match all sessions with the specified `DisplayName` or `Process.name()`. An integer will match only the audio session with the specified `ProcessId`. A list of sessions instances can be retrieved by `GetAllSessions()`.

CHAPTER 4

Support

- You can ask programming questions on stackoverflow, using the `[audiomath]` tag (https://stackoverflow.com/questions/tagged/audiomath)
- If an issue is confirmed to be a technical problem or bug, you can submit details at https://bitbucket.org/snapproject/audiomath-gitrepo/issues using the "Create Issue" button. Remember to include full details of any error, as well as `audiomath.ReportVersions()` output if possible.

CHAPTER 5

License

This file is part of the audiomath project, a Python package for recording, manipulating and playing sound files.

The audiomath distribution includes binaries from the third-party AVbin and PortAudio projects, released under their own licenses. See the respective copyright, licensing and disclaimer information for these projects in the subdirectories `audiomath/_wrap_avbin` and `audiomath/_wrap_portaudio` . It also includes a fork of the third-party Python package `pycaw`, released under its original license (see `audiomath/pycaw_fork.py`).

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

audiomath, 19
audiomath.PortAudioInterface, 48
audiomath.PsychToolboxInterface, 52
audiomath.Signal, 56
audiomath.SystemVolume, 60

Index

A

add() (*audiomath.Queue method*), 39
Amplitude() (*audiomath.Sound method*), 24
ap2fft() (*in module audiomath.Signal*), 59
append() (*audiomath.Queue method*), 39
ApplyWindow() (*audiomath.Sound method*), 24
ApplyWindow() (*in module audiomath.Signal*), 57
audiomath (*module*), 19
audiomath.PortAudioInterface (*module*), 48
audiomath.PsychToolboxInterface (*module*), 52
audiomath.Signal (*module*), 56
audiomath.SystemVolume (*module*), 60
autoAdvance (*audiomath.Player attribute*), 35
AutoScale() (*audiomath.Sound method*), 24

B

Back() (*audiomath.Queue method*), 38
bits (*audiomath.Sound attribute*), 29
Bulk() (*audiomath.Sound method*), 24
bytes (*audiomath.Sound attribute*), 29

C

Cat() (*audiomath.Sound method*), 24
Center() (*audiomath.Sound method*), 24
clear() (*audiomath.Queue method*), 39
Click() (*in module audiomath.Signal*), 58
Concatenate() (*audiomath.Sound method*), 24
Concatenate() (*in module audiomath*), 20
Copy() (*audiomath.Sound method*), 25
copy() (*audiomath.Sound method*), 29
currentSound (*audiomath.Queue attribute*), 40
Cut() (*audiomath.Recorder method*), 41
Cut() (*audiomath.Sound method*), 25

D

dat2str() (*audiomath.Sound method*), 29
Delay (*class in audiomath*), 41
Detect() (*audiomath.Sound method*), 25
duration (*audiomath.Sound attribute*), 29
Duration() (*audiomath.Sound method*), 25
dynamics (*audiomath.Player attribute*), 35

E

Envelope() (*audiomath.Sound method*), 25
extend() (*audiomath.Queue method*), 39

F

Fade() (*audiomath.Sound method*), 26
Fader (*class in audiomath*), 40
ffmpeg (*class in audiomath*), 44
ffmpeg.NotInstalled, 45
fft2ap() (*in module audiomath.Signal*), 58
fftfreqs() (*in module audiomath.Signal*), 58
FindDevice() (*in module audiomath.PortAudioInterface*), 49
FindDevice() (*in module audiomath.PsychToolboxInterface*), 53
FindDevices() (*in module audiomath.PortAudioInterface*), 49
FindDevices() (*in module audiomath.PsychToolboxInterface*), 53
Forward() (*audiomath.Queue method*), 39
fs (*audiomath.Sound attribute*), 29
func() (*audiomath.Synth static method*), 31

G

GenerateWaveform() (*audiomath.Sound method*), 26
GenerateWaveform() (*in module audiomath.Signal*), 57
GetDeviceInfo() (*in module audiomath.PortAudioInterface*), 49
GetDeviceInfo() (*in module audiomath.PsychToolboxInterface*), 53
GetDynamic() (*audiomath.Player method*), 32
GetHostApiInfo() (*in module audiomath.PortAudioInterface*), 49

GetHostApiInfo() (*in module audiomath.PsychToolboxInterface*), 53
GetInputDevice() (*in module audiomath.PortAudioInterface*), 51
GetOutputDevice() (*in module audiomath.PortAudioInterface*), 50
GetVolume() (*in module audiomath.SystemVolume*), 60
GrokFormat() (*audiomath.ffmpeg static method*), 45
GrokFormat() (*audiomath.sox static method*), 47

H

head (*audiomath.Player attribute*), 36
head (*audiomath.Recorder attribute*), 43
Hilbert() (*in module audiomath.Signal*), 59

I

index() (*audiomath.Queue method*), 39
insert() (*audiomath.Queue method*), 40
Install() (*audiomath.ffmpeg class method*), 45
Install() (*audiomath.sox class method*), 47
InverseSpectrum() (*in module audiomath.Signal*), 59
IsInstalled() (*audiomath.ffmpeg class method*), 45
IsInstalled() (*audiomath.sox class method*), 47
IsolateChannels() (*audiomath.Sound method*), 26

L

LATENCY_CLASS (*class in audiomath.PsychToolboxInterface*), 54
Left() (*audiomath.Sound method*), 27
levels (*audiomath.Player attribute*), 36
levels (*audiomath.PsychToolboxInterface.Player attribute*), 55
loop (*audiomath.Player attribute*), 36
loop (*audiomath.Recorder attribute*), 43
LowLatencyMode() (*in module audiomath.PortAudioInterface*), 48

M

MakeFall() (*in module audiomath*), 20
MakeHannWindow() (*audiomath.Sound method*), 27
MakeHannWindow() (*in module audiomath*), 20
MakePlateau() (*in module audiomath*), 20
MakeRecording() (*audiomath.PsychToolboxInterface.Recorder class method*), 56
MakeRecording() (*audiomath.Recorder class method*), 41
MakeRise() (*in module audiomath*), 20
MixDownToMono() (*audiomath.Sound method*), 27
ModulateAmplitude() (*audiomath.Sound method*), 27
ModulateAmplitude() (*in module audiomath.Signal*), 57
MoveSound() (*audiomath.Queue method*), 39
msec2samples() (*in module audiomath.Signal*), 56
muted (*audiomath.Player attribute*), 36
muted (*audiomath.PsychToolboxInterface.Player attribute*), 55

N

nbits (*audiomath.Sound attribute*), 30
nbytes (*audiomath.Sound attribute*), 30
nchan (*audiomath.Sound attribute*), 30
nChannels (*audiomath.Sound attribute*), 29
NextTrack() (*audiomath.Player method*), 32
Noise() (*in module audiomath.Signal*), 58
norm (*audiomath.Player attribute*), 36
norm (*audiomath.PsychToolboxInterface.Player attribute*), 55
nsamp (*audiomath.Sound attribute*), 30
nSamples (*audiomath.Sound attribute*), 30
numberOfChannels (*audiomath.Sound attribute*), 30
NumberOfChannels() (*audiomath.Sound method*), 27
numberOfSamples (*audiomath.Sound attribute*), 30
NumberOfSamples() (*audiomath.Sound method*), 27

P

PadEndTo() (*audiomath.Sound method*), 27
PadStartTo() (*audiomath.Sound method*), 27
pan (*audiomath.Player attribute*), 36
pan (*audiomath.PsychToolboxInterface.Player attribute*), 55
Pause() (*audiomath.Player method*), 33
Pause() (*audiomath.Recorder method*), 42
PitchShift (*audiomath.Sound attribute*), 29
Play() (*audiomath.Player method*), 33
Play() (*audiomath.Queue method*), 39
Play() (*audiomath.Sound method*), 27
Player (*class in audiomath*), 31
Player (*class in audiomath.PortAudioInterface*), 50
Player (*class in audiomath.PsychToolboxInterface*), 54
playing (*audiomath.Player attribute*), 36
Plot() (*audiomath.Sound method*), 27
PlotSignal() (*in module audiomath.Signal*), 60
PlotSpectrum() (*in module audiomath.Signal*), 60
pop() (*audiomath.Queue method*), 40
position (*audiomath.Queue attribute*), 40
PreviousTrack() (*audiomath.Player method*), 33
Process() (*audiomath.ffmpeg class method*), 45
Process() (*audiomath.sox class method*), 47

Q

queue (*audiomath.Player attribute*), 36
Queue (*class in audiomath*), 38

QueueTest() (*in module audiomath*), 20

R

Read() (*audiomath.Sound method*), 28
ReadSamples() (*audiomath.Recorder method*), 42
Reconstruct() (*in module audiomath.Signal*), 59
Record() (*audiomath.Recorder method*), 42
Record() (*in module audiomath*), 20
Recorder (*class in audiomath*), 41
Recorder (*class in audiomath.PortAudioInterface*), 51
Recorder (*class in audiomath.PsychToolboxInterface*), 56
remove() (*audiomath.Queue method*), 40
repeatQueue (*audiomath.Player attribute*), 37
Resample() (*audiomath.Player method*), 33
Resample() (*audiomath.Sound method*), 28
ResetDynamics() (*audiomath.Player method*), 33
Restart() (*audiomath.Queue method*), 39
Reverse() (*audiomath.Sound method*), 28
ReweightSpectrum() (*in module audiomath.Signal*), 59
Right() (*audiomath.Sound method*), 28
rms (*audiomath.Sound attribute*), 30

S

samples2msec() (*in module audiomath.Signal*), 56
SamplesToSeconds() (*audiomath.Sound method*), 28
SawtoothWave() (*in module audiomath.Signal*), 58
Seconds() (*in module audiomath.PortAudioInterface*), 50
Seconds() (*in module audiomath.PsychToolboxInterface*), 56
SecondsToSamples() (*audiomath.Sound method*), 29
Seek() (*audiomath.Player method*), 33
Seek() (*audiomath.Recorder method*), 43
Set() (*audiomath.Player method*), 34
Set() (*audiomath.Recorder method*), 43
SetDefaultVerbosity() (*in module audiomath.PortAudioInterface*), 48
SetDynamic() (*audiomath.Player method*), 34
SetInputDevice() (*in module audiomath.PortAudioInterface*), 51
SetOutputDevice() (*in module audiomath.PortAudioInterface*), 50
SetPosition() (*audiomath.Queue method*), 39
SetVolume() (*in module audiomath.SystemVolume*), 60
Shoulder() (*in module audiomath.Signal*), 59
SineWave() (*in module audiomath.Signal*), 58
sound (*audiomath.Player attribute*), 37
sound (*audiomath.PsychToolboxInterface.Player attribute*), 56
Sound (*class in audiomath*), 22
sox (*class in audiomath*), 46
sox.NotInstalled, 47
Spectrum() (*in module audiomath.Signal*), 59
speed (*audiomath.Player attribute*), 37
SplitChannels() (*audiomath.Sound method*), 29
SquareWave() (*in module audiomath.Signal*), 58
Stack() (*audiomath.Sound method*), 29
Stack() (*in module audiomath*), 21
Start() (*audiomath.Recorder method*), 43
Stop() (*audiomath.Player method*), 34
Stop() (*audiomath.Recorder method*), 43
Stream (*class in audiomath.PortAudioInterface*), 50
Stream (*class in audiomath.PsychToolboxInterface*), 54
SwapSounds() (*audiomath.Queue method*), 39
SwitchToTrack() (*audiomath.Player method*), 34
synchronizeDynamics (*audiomath.Player attribute*), 37
Synth (*class in audiomath*), 30
SystemVolumeSetting (*class in audiomath.SystemVolume*), 60

T

Tabulate() (*in module audiomath.PortAudioInterface*), 49
TestSound() (*in module audiomath*), 21
Timebase() (*in module audiomath.Signal*), 60
TimeStretch (*audiomath.Sound attribute*), 29
ToneTest() (*in module audiomath*), 22
Toy() (*in module audiomath.Signal*), 59
track (*audiomath.Player attribute*), 38
TriangleWave() (*in module audiomath.Signal*), 58
Trim() (*audiomath.Sound method*), 29

U

UnwrapDiff() (*in module audiomath.Signal*), 56

V

vol (*audiomath.Player attribute*), 38
vol (*audiomath.PsychToolboxInterface.Player attribute*), 56
volume (*audiomath.Player attribute*), 38
volume (*audiomath.PsychToolboxInterface.Player attribute*), 56

W

Wait() (*audiomath.Player method*), 35
Wait() (*audiomath.Recorder method*), 43
WaitFor() (*audiomath.Player method*), 35
WaitFor() (*audiomath.Recorder method*), 43
with_traceback() (*audiomath.ffmpeg.NotInstalled method*), 45
with_traceback() (*audiomath.sox.NotInstalled method*), 47

Y

y (*audiomath.Sound attribute*), 30