chaudio Documentation

Release 0.0.2

Cade Brown

Feb 02, 2018

Contents:

1 Installing **3**

2 Getting Started **5**

3 API Reference **7**

3.1 Programmatic Music Synthesis (`chaudio`) 7

3.1.1 chaudio.arrangers 7

3.1.2 chaudio.instruments 10

3.1.3 chaudio.io 15

3.1.4 chaudio.plugins 17

3.1.5 chaudio.source 21

3.1.6 chaudio.util 26

3.1.7 chaudio.waves 29

4 Indices and tables **35**

Python Module Index **37**

chaudio is a python library for making music. Completely created through python source code, you can create virtually any music you can imagine.

This is the online user documentation for chaudio. You can see all the source code for chaudio on ChemicalDevelopment's GitHub: chaudio on GitHub

CHAPTER 1

Installing

chaudio is available as a pip package.

python3 is suggested, but it is python2 compatable.

So, either:

```
pip3 install chaudio
```

Or, for python2 installation:

```
pip install chaudio
```

Note that this requires scipy and numpy to be installed, which sometimes requires system libraries.

To install on a Debian based system (such as Ubuntu), first run:

```
sudo apt-get install python3-scipy
```

Or, for python2 installation:

```
sudo apt-get install python-scipy
```

Once you've ran this, rerun the pip installation command for chaudio

CHAPTER 2

Getting Started

Once you've installed chaudio, you will want to start making music.

First, in either python2 or python3, you can run:

```
import chaudio
```

And now you have access to all of the chaudio library

First, we'll start with generating a simple note:

```
# create our array of time samples (lasting 5 seconds)
t = chaudio.times(5)
```

The `chaudio.times` function returns an array of times at which audio samples are to be created, for 5 seconds of data. By default, the samplerate (also referred to as *hz*, or samples per second) is 44100, which is the most common value.

```
pitch = chaudio.note("A3")
```

The `chaudio.note` function returns a pitch (in *hz*) of the desired note and octave. For example, `chaudio.note("A3") == 220.0`

```
# our air pressure array, generated using a square wave
y = chaudio.waves.square(t, pitch)
```

The `chaudio.waves.square` is a waveform function, that is, it takes time sample values, and a frequency, and returns a signal representing that (in this case, the form of this function is called a square wave). Each waveform has its own timbre (pronounced *TAM-BER*), which we will cover in a future section.

Now, we use our sample times, `t`, and our note pitch `pitch`, feed that through our waveform generator, `chaudio.waves.square`, which returns an array of pressure values. And, viola, we have the data representing an *A3* played on a square oscillator. But, how do we hear it?

```
chaudio.tofile("~/Music/square_A3.wav", y)
```

The function `chaudio.tofile` takes either a filename, or file pointer object, and then a data array of samples. It does all neccessary conversions, and now, just open up your music folder `~/Music`, and open `square_A3.wav`. You should hear the square wave playing the *A3* note.

So, all together, this example can be ran as such:

```
import chaudio

t = chaudio.times(5)

pitch = chaudio.note("A3")

y = chaudio.waves.square(t, pitch)

chaudio.tofile("~/Music/square_A3.wav", y)
```

CHAPTER 3

API Reference

Here' the API reference for chaudio,

3.1 Programmatic Music Synthesis (`chaudio`)

Submodules:

arrangers	Arrangers (*chaudio.arrangers*)
instruments	Instruments (*chaudio.instruments*)
io	Input/Output functionality (*chaudio.io*)
plugins	Plugin Audio Processing (*chaudio.plugins*)
source	Audio Source (*chaudio.source*)
util	Utility Functions (*chaudio.util*)
waves	Waveform Generation Functions (*chaudio.waves*)

3.1.1 chaudio.arrangers

Arrangers (`chaudio.arrangers`)

Can record inputs, and detect if they change, and regenerate their source automatically

This is essentially "smart" JIT audio production

An Arranger can have another Arranger inserted, and it will keep track of whether the sources have changed, only recalculating if needed.

Classes

Arranger(**kwargs)	this class is for combining sounds together, given a point, and applying plugins:
ExtendedArranger(**kwargs)	Extends the basic arranger (*chaudio.arrangers.Arranger*)
InsertCall(key, val, kwargs)	Data structure to have handle each time something is inserted, so that the action can be reconstructed later

class chaudio.arrangers.**InsertCall**(*key*, *val*, *kwargs*)

Data structure to have handle each time something is inserted, so that the action can be reconstructed later

__init__(*key*, *val*, *kwargs*)

Initializes an insert call

Parameters

- **key** (*obj*) – Key used to input
- **val** (*obj*) – Value given to set at key
- **kwargs** (*dict*) – Which arguments were given with the insertion

Returns The result representing an insert call

Return type *chaudio.arrangers.InsertCall*

__weakref__

list of weak references to the object (if defined)

class chaudio.arrangers.**Arranger**(***kwargs*)

this class is for combining sounds together, given a point, and applying plugins:

Only has support for inputting at a number of samples, serves as a base class.

chaudio.arrangers.ExtendedArranger is probably the best for most users

__init__(***kwargs*)

Initializes an arranger

Parameters **kwargs** (*(keyword arguments)*) – Which arguments were given and should be used in operations. Essentially config options.

Returns The result representing an insert call

Return type *chaudio.arrangers.InsertCall*

getarg(*key*, *default=None*)

Returns the value stored in key word arguments, or a default if it is not contained

Parameters

- **key** (*obj*) – Key, normally a str
- **default** (*obj*) – What to return if the key word arguments does not contain the specified key

Returns The value stored in key word arguments, or a default if it is not contained

Return type obj

setarg(*key*, *val*, *replace=True*)

Sets the argument, specifying whether or not to override

Parameters

- **key** (*obj*) – Key, normally a str

- **val** (*obj*) – value to store
- **replace** (*bool*) – If True, replace if `key` is already contained. If not, only replace if the key is not in the key word arguments

add_insert_plugin(*plugin*)
Adds a plugin that is applied when a clip or value is inserted

See *chaudio.plugins* for some plugins, and a description.

Essentially, `plugin` is added to the chain (at the end), which processes the output of the previous plugin (or it is the first, in which case it acts on the data inserted).

Parameters **plugin** (`chaudio.plugins.Basic`) – What plugin to add

Returns index of the plugin, such that `plugin == arranger.insert_plugins[RETURN]` where `RETURN` is the return value of this function.

Return type int

add_final_plugin(*plugin*)
Adds a plugin that is applied to the entire arranger's computed things.

See *chaudio.plugins* for some plugins, and a description.

Essentially, `plugin` is added to the chain (at the end), which processes the output of the previous plugin (or it is the first, in which case it acts on the data inserted).

First, each insert has all the insert plugins applied (see method *chaudio.arrangers.Arranger.add_insert_plugin()*), and all of the inserted sources are combined and inserted at their respective places, and all the final plugins are applied.

Parameters **plugin** (`chaudio.plugins.Basic`) – What plugin to add

Returns index of the plugin, such that `plugin == arranger.final_plugins[RETURN]` where `RETURN` is the return value of this function.

Return type int

apply_insert(*insert_call*)
Internal method to apply an insert call object

Essentially change self._source by applying an insert. This method exists so that inserted values can be changed, and internal references can be updated if any change.

Parameters **insert_call** (*chaudio.arrangers.InsertCall*) – What data structure to insert

insert_sample(*sample*, *_data*, ***kwargs*)
Inserts data at an offset, in sample measurements

Add a insert value which adds `_data` at the offset `sample`. Note that this doesn't take into account the time in seconds, for that use *chaudio.arrangers.ExtendedArranger*, specifically the method *chaudio.arrangers.ExtendedArranger.insert_time()*

Parameters **insert_call** (*chaudio.arrangers.InsertCall*) – What data structure to insert

__weakref__
list of weak references to the object (if defined)

class chaudio.arrangers.**ExtendedArranger**(***kwargs*)
Extends the basic arranger (*chaudio.arrangers.Arranger*)

__init__(***kwargs*)
Initializes the ExtendedArranger.

Add a insert value which adds `_data` at the offset `sample`. Note that this doesn't take into account the time in seconds, for that use *chaudio.arrangers.ExtendedArranger*, specifically the method *chaudio.arrangers.ExtendedArranger.insert_time()*

Keyword arguments:

Hz int, the samplerate

Timesignature *chaudio.util.TimeSignature*, which is the time signature used. The default is 4/4 in 60 bpm (so that 1 beat == 1 second)

Setitem either "sample", "time", or "beat". This controls how the `chaudio.arrangers.ExtendedArranger.__setitem__()` functionality works (when you call it like *extarranger[X] = Y*.

if `setitem == "sample"`, the `X` is treated as the sample (and the behaviour is the same as *chaudio.arrangers.Arranger*). This is the default.

if `setitem == "time"`, the `X` is treated as the time in seconds

if `setitem == "beat"`, the `X` is treated as a number of beats, and is used in accordance with `timesignature` (see above). Or, it can be a tuple of `measures, beats`. For example, `extarranger[M, B] = Y` can be used.

Parameters **kwargs** (*(key word arguments)*) – Extended arranger adds `hz`, `timesignature`, and `setitem` as usage values. See the description above for an explanation of these.

insert_time(*t*, *_data*)
Inserts audio data at a specified time (in seconds)

Parameters

- **t** (*int, float*) – Time, in seconds, to apply the audio at
- **_data** (list, tuple, np.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is the data (which can be any generic format, the internal classes will figure out how to make it work out).

insert_beat(*beat*, *_data*)
Inserts audio data at a specified time (in beats, or in (measures, beats) format)

Parameters

- **beat** (*int, float, tuple*) – The number of beats, or a tuple containing `measures, beats`. This is converted to time, which *chaudio.arrangers.ExtendedArranger.insert_time()* is called internally.
- **_data** (list, tuple, np.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is the data (which can be any generic format, the internal classes will figure out how to make it work out).

3.1.2 chaudio.instruments

Instruments (`chaudio.instruments`)

Support for instrument plugins that can play notes, MIDI data, and other formats

Classes

ADSREnvelope([A, D, S, R])	This is an ADSR envelope.
Instrument(**kwargs)	Base class to extend if you have an instrument
LFO([waveform, hz, tweak, amp, dc_shift, ...])	
MultiOscillator(osc, **kwargs)	Similar to LMMS's triple oscillator (which itself was based on minimoog synth), but with a variable number
MultiSampler([sampler_dict])	
Oscillator([waveform, amp, amp_env, ...])	Represents a basic oscillator, which is the base class for most synths
Sampler(**kwargs)	

class chaudio.instruments.**Instrument**(***kwargs*)

Base class to extend if you have an instrument

__init__(***kwargs*)

Initializes an Instrument (which is a base class that should not be used, please use *chaudio.instruments.Oscillator* or another class!)

Parameters ****kwargs** (*(key word arguments)*) – The generic instrument arguments

Returns A generic instrument object

Return type *chaudio.instruments.Instrument*

raw_note(***kwargs*)

Returns raw note source (i.e. without plugins added)

MOST PEOPLE SHOULD NOT NEED THIS FUNCTION!

Please use the *chaudio.instruments.Instrument.note()* function for external needs, as this method is used internally there, and then plugins are applied.

This method is not implemented in this base (i.e. *chaudio.instruments.Instrument*) class, but should be implemented by any actual instrument.

Parameters ****kwargs** (*(key word arguments)*) – The generic instrument arguments

Returns A source representing the raw note of the instrument (without plugins).

Return type chaudio.source.Stereo

note(***kwargs*)

Returns raw note source with all plugins applied

The *freq* argument is specified as the actual note being played. So, to play an A, use `instrument.note(freq="A", ...)`

Parameters ****kwargs** (*(key word arguments)*) – The generic instrument arguments (use *freq* for the note value)

Returns A source representing instrument playing a note

Return type chaudio.source.Stereo

add_plugin(*plugin*)

Adds a processing plugin

Parameters **plugin** (chaudio.plugins.Basic) – Plugin object that extends Basic

remove_plugin(*plugin*)

Removes a plugin, by the plugin object, or the index

Parameters **plugin** (chaudio.plugins.Basic or int) – Plugin object that extends Basic, or index

copy()
Returns a copy of the object

Returns A copy of the object. Keeps the same type, however

Return type *chaudio.instruments.Instrument* (or whatever class the object is)

merged_kwargs(*specific_kwargs*, *exclude=[]*)
Returns the merged kwargs (i.e. the input replaces values not specified as defaults.) and then remove exclude vals.

Parameters

- **specific_kwargs** (*dict*) – What was passed to the specific function (like *chaudio.instruments.Instrument.note()*) that needs to override the initialiezd kwargs.
- **exclude** (*list, tuple*) – Which arguments to remove (i.e. exclude) from the result

Returns The merged results, with anything from specific_kwargs taking precedence over the defaults, then remove all from exclude

Return type dict

__getitem__(*key*)
Returns the configuration/kwargs value at a specified key

Parameters **key** (*obj*) – Whatever key the value was stored as (typically str)

Returns The value stored as a kwarg

Return type obj

__setitem__(*key*, *val*)
Sets the configuration/kwargs value at a specified key to a provided value

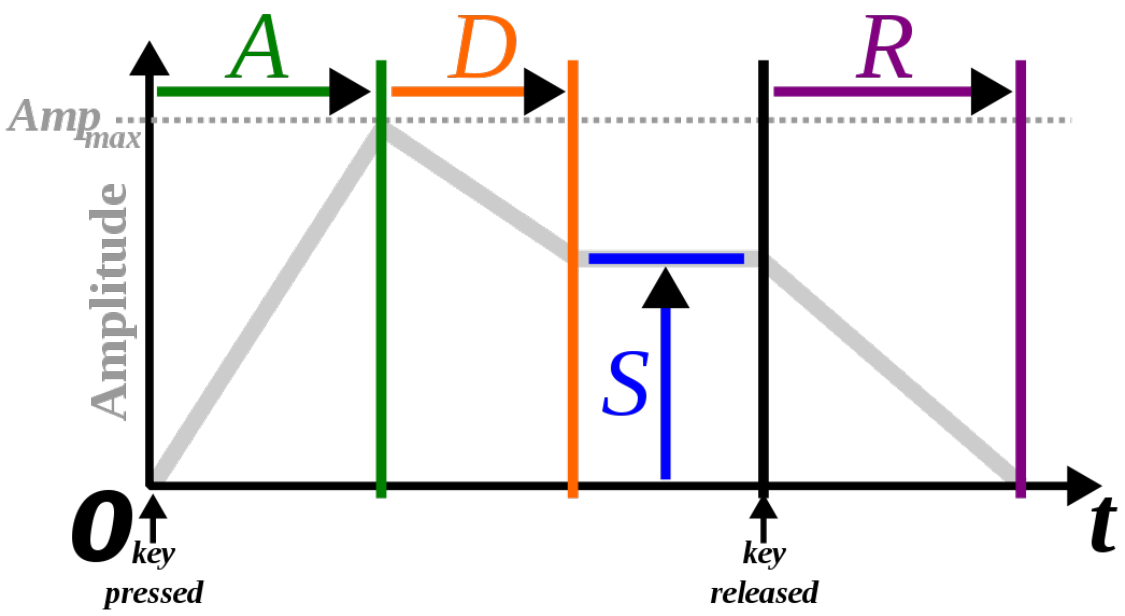
Parameters

- **key** (*obj*) – Whatever key the value was stored as (typically str)
- **val** (*obj*) – What value to set at key

__weakref__
list of weak references to the object (if defined)

class chaudio.instruments.**ADSREnvelope**(*A=0*, *D=0*, *S=1*, *R=0*)
This is an ADSR envelope.

For a good explanation, see ADSR Envelope one wikiaudio.



__init__(*A=0*, *D=0*, *S=1*, *R=0*)

Initializes the envelope with common parameters.

Parameters

- **A** (*float*) – 'attack' value, in seconds. Essentially the envelope is ramping up until `A` seconds
- **D** (*float*) – 'decay' value, in seconds. The envelope will scale gently to `S` between time `A` and `A + D` seconds.
- **S** (*float*) – 'sustain' value, as an amplitude between 0.0 and 1.0. This is the value that the envelope holds between `A + D` and `t - R` seconds (where `t` is the length of the segment that is being enveloped).
- **R** (*float*) – 'release', value in seconds. This is the time (from the end of the time values being enveloped) that the ADSR envelope starts fading out.

calc_val(*t*, ***kwargs*)

Returns the envelope values for a time sample array.

This returns the values (which are all 0.0 to 1.0) of the envelope, applied over the times given in `t`. The result has the same length as t, so that you can apply operations to `t` and others. See the examples below. This is used in *chaudio.instruments.Oscillator*.

Parameters

- **t** (*np.ndarray*) – The value of time samples. These are generated (typically) using the *chaudio.util.times()* method.
- **kwargs** (*(key word args)*) – These can override `A`, `D`, `S`, or `R` for a specific call to this function without forever replacing the defaults.

Returns A result with the same length as `t` (so you can do operations with them).

Return type np.ndarray

Examples

```
>>> t = chaudio.util.times(4)
>>> wave = chaudio.waves.triangle(t, hz=220)
>>> env = chaudio.instruments.ADSREnvelope(A=.4, D=1.0, S=.4, R=.8)
>>> y = wave * env.calc_val(t)
>>> # y now contains the wave with the envelope value
>>> chaudio.play(y)
```

__weakref__
list of weak references to the object (if defined)

class chaudio.instruments.**Oscillator**(*waveform=<function sin>*, *amp=1.0*, *amp_env=<chaudio.instruments.ADSREnvelope object>*, *amp_lfo=<chaudio.instruments.LFO object>*, *samplerate=None*, *phase_shift=0*, *freq_shift=0*, *freq_lfo=<chaudio.instruments.LFO object>*, *tweak=None*, *tweak_lfo=<chaudio.instruments.LFO object>*, *pan=0*, ***kwargs*)

Represents a basic oscillator, which is the base class for most synths

__init__(*waveform=<function sin>*, *amp=1.0*, *amp_env=<chaudio.instruments.ADSREnvelope object>*, *amp_lfo=<chaudio.instruments.LFO object>*, *samplerate=None*, *phase_shift=0*, *freq_shift=0*, *freq_lfo=<chaudio.instruments.LFO object>*, *tweak=None*, *tweak_lfo=<chaudio.instruments.LFO object>*, *pan=0*, ***kwargs*)

Initializes an oscillator, given waveform and a host of other parameters

Keep in all parameters can be overriden in individual calls to the `chaudio.instruments.Oscillator.note()` function. So, to override the `phase_shift` for a single note, run like ‘‘osc.note(“A4”, phase_shift=2) to temporarily override the initialized parameter.

To change the values you initialized with, set like so: `osc["phase_shift"] = 2`

Parameters that accept a `tuple` and `dict` (such as `phase_shift`) mean that it can accept left and right values that differ. So, the left channel has a different phase offset than the right side. If you give a tuple for these values, `v[0]` is for the left and `v[1]` is for the right. If given a dict, `v["left"]` is for the left and `v["right"]` is for the right. And remember, all parameters accept a single value as a float/int/None, in which the left and right values are both taken as `v`.

Parameters

- **waveform** (*func*) – What internal waveform to generate sounds based on. See module *chaudio.waves* for a list of defaults included with chaudio, as well as their function.
- **amp** (*float, int*) – the amplitude of the waveform. This is useful when combining multiple oscillators (see *chaudio.instruments.MultiOscillator* for example on this).
- **samplerate** (*int, None*) – What is the samplerate that should be used
- **phase_shift** (*float, tuple, dict*) – The offset in the wavefunction. A phase shift of 0 means use the default waveform function. A phase shift of .5 means begin halfway through the first oscillation.
- **freq_shift** (*float, tuple, dict*) – The offset, in cents, that the oscillator transposes given notes to. Essentially, if given `freq` to play, the returned source containing data is the note `freq` transposed `freq_shift` cents.
- **tweak** (*float, None, tuple, dict*) – The tweak value to apply on the waveform.

- **pan** (*float, None*) – A panning (Left/Right) value to change the offset in the stereo space. -1.0 representing all the way left, +1.0 representing all the way right.

Returns The instrument object

Return type *chaudio.instruments.Oscillator*

raw_note(***kwargs*)

Returns the result of the instrument performing a note for specified parameters

Basic usage is `osc.note(freq="A4", amp=.5, ...)` and that overrides the `amp` value set on creation.

You can permanently update these values with `osc["amp"] = .5` to update the default used if nothing is passed into the note function.

Parameters

- **freq** (*int, float, str, np.ndarray*) – This can be a frequency directly, or a string of a note name (see *chaudio.util.note()* for what is suppoerted). Additionally, it can be an array of frequencies at time values. Note that this should contain data with the sameplerate as the oscillator. You can check the oscillator sample rate with: `osc["samplerate"]`. As a consequence, it also needs to be the same shape as the time parameter `t` generated array
- **kwargs** (*(key word args)*) – These are all values that can override the default values (which all are documented in the *chaudio.instruments.Oscillator.__init__()* method).

Returns The source representing the oscillator playing the note

Return type `chaudio.source.Stereo`

class chaudio.instruments.**MultiOscillator**(*osc*, ***kwargs*)

Similar to LMMS's triple oscillator (which itself was based on minimoog synth), but with a variable number

__init__(*osc*, ***kwargs*)

Returns an instrument multiplexor with oscillators

Parameters

- **osc** (*list, tuple, None*) – The group of oscillators. These are all the oscillators that are played each time you ask for a note. You can change oscillators after construction using the `chaudio.instruments.MultiOscillator.add_osc()` and `chaudio.instruments.MultiOscillator.remove_osc()` methods.
- **kwargs** (*(key word args)*) – These are all values that can override the default values (which all are documented in the *chaudio.instruments.Oscillator.__init__()* method).

Returns The multioscillator representing the oscillators playing the note

Return type *chaudio.instruments.MultiOscillator*

3.1.3 chaudio.io

Input/Output functionality (`chaudio.io`)

Allows any data type to be stored to a file, returned as a string, read from a file or string, and other I/O issues.

At this point, only WAVE integer formats are accepted, so 8 bit, 16 bit, 24 bit, and 32 bit WAVE formats all work.

WAVE 32f format does not work.

In the future, support for `.ogg` and `.mp3` files will be hopefully added.

Functions

fromfile(filename[, silent])	Returns file contents of a WAVE file (either name or file pointer) as a *chaudio.source.Source*
fromstring(strdata, *args, **kwargs)	Treat the input as WAVE file contents, and return a *chaudio.source.Source*.
play(_audio[, waveformat])	Plays the audio through the system speaker
tofile(filename, _audio[, waveformat, ...])	Output some sort of audio to a file (which can be a name or file pointer).
tostring(_audio, *args, **kwargs)	Returns the WAVE file contents.

Classes

`WaveFormat`(name, dtype, samplewidth, ...)	

chaudio.io.**play**(*_audio*, *waveformat='16i'*)

Plays the audio through the system speaker

Requires simpleaudio `pip3 install simpleaudio` to work (which is a dependency of chaudio)

Parameters

- **_audio** (*chaudio.source.Source*, *chaudio.arrangers.Arranger*, np.ndarray) – This is converted to *chaudio.source.Source*, and then output.
- **waveformat** (str, `chaudio.io.WaveFormat`) – This describes how to convert the data. Should probably be an integer format, and the default is good enough for anyone. See `chaudio.io.WaveFormat` for how to use it.

Returns **'simpleaudio.PlayObject <http** – You can use this to cancel or change playback

Return type //simpleaudio.readthedocs.io/en/latest/simpleaudio.html#simpleaudio.PlayObject>'_

chaudio.io.**fromfile**(*filename*, *silent=False*)

Returns file contents of a WAVE file (either name or file pointer) as a *chaudio.source.Source*

Note that they are not "normalized" as in using *chaudio.util.normalize()*, but rather simply converted from the internal WAVE formats (which are integers), and divided by the maximum integer of that size. That way, all WAVE formats will return (within rounding) the same result when called with this function, so the original volume is conserved. This is the behaviour audacity has when reading files, which is to convert to 32f format internally.

This supports all standard WAVE integer formats, 8 bit, 16 bit, 24 bit, and 32 bit.

Note that WAVE 32f format is **NOT** supported yet

Parameters

- **filename** (*str, file*) – If a string, that file is opened, or if it is a file object already (which can be an io.StringIO object), that is used instead of opening another.
- **silent** (*bool*) – Print out what file is being used, so that the user knows what's happening

Returns A chaudio Source class, with appropriate channels, samplerate, and a dtype of float32

Return type *chaudio.source.Source*

chaudio.io.**fromstring**(*strdata*, **args*, ***kwargs*)
Treat the input as WAVE file contents, and return a *chaudio.source.Source*.

Parameters **strdata** (*str*) – Treat strdata as the WAVE file contents

Returns A chaudio Source class, with appropriate channels, samplerate, and a dtype of float32

Return type *chaudio.source.Source*

chaudio.io.**tofile**(*filename*, *_audio*, *waveformat='16i'*, *normalize=True*, *silent=False*)
Output some sort of audio to a file (which can be a name or file pointer).

Always to WAVE format, and specify waveformat in order to change what kind. Default, it is 16 bit integer (CD quality).

Parameters

- **filename** (*str, file*) – If a string, that file is opened, or if it is a file object already (which can be an io.StringIO object), that is used instead of opening another.
- **_audio** (np.ndarray, *chaudio.source.Source*, chaudio.source.Arranger) – Casts _audio to a *chaudio.source.Source*, which will work on any chaudio type. You shouldn't have to worry about this, it stays truthful to the input.
- **waveformat** (*{ '8i', '16i', '24i', '32i' }*) – Describes the number of bits per sample, and what type of data to write ('i' is integer).
- **normalize** (*bool*) – Whether or not to normalize before writing. This should be the default, to avoid any clipping.
- **silent** (*bool*) – Print out what file is being used, so that the user knows what's happening

chaudio.io.**tostring**(*_audio*, **args*, ***kwargs*)
Returns the WAVE file contents. Essentially returns what *chaudio.io.tofile()* would have written to a file.

Parameters

- **_audio** (np.ndarray, *chaudio.source.Source*, chaudio.source.Arranger) – Casts _audio to a *chaudio.source.Source*, which will work on any chaudio type. You shouldn't have to worry about this, it stays truthful to the input.
- **waveformat** (*{ '8i', '16i', '24i', '32i' }*) – Describes the number of bits per sample, and what type of data to write ('i' is integer).
- **normalize** (*bool*) – Whether or not to normalize before writing. This should be the default, to avoid any clipping.

Returns An str representing the WAVE file contents.

Return type str

3.1.4 chaudio.plugins

Plugin Audio Processing (chaudio.plugins)

These are plugins that take in an input, perform some action on it to alter the sound, and then return the result as a *chaudio.source.Source*.

Some common ones:

echo	Adds in the echo effect, with each successive echo being decayed.
fade	A gentle fade in and out
filters	filters to remove frequency ranges, pass zones, bands, etc
noise	adds white noise to input
resolution	changes the minimum resolution
volume	A simple multiplier to scale the volume

chaudio.plugins.echo

Adds in the echo effect, with each successive echo being decayed.

Classes

Echo(**kwargs)	Adds in the echo effect, with each successive echo being decayed.

class chaudio.plugins.echo.**Echo**(***kwargs*)

Adds in the echo effect, with each successive echo being decayed.

process(*_data*)

Returns the result, but echoed

So, the the amplitude of the `n` th echo is `kwargs["amp"] * (n) ** kwargs["decay"]`

"idelay" The initial delay, in seconds, before the echos begin at all

"delay" The delay for each successive echo

"num" How many echos to factor in

"amp" The base amplitude of all echos

"decay" The multiplication of the signal each successive echo

chaudio.plugins.fade

A gentle fade in and out

Classes

Fade(**kwargs)	A gentle fade in and out

class chaudio.plugins.fade.**Fade**(***kwargs*)

A gentle fade in and out

process(*_data*)

Returns the result, but faded given a few parameters

"fadein" `True` to fade at the beginning, `False` to not

"fadeout" `True` to fade at the end, `False` to not

"sec" the length, in seconds, of how long to fade

Essentially for the first samples til `sec` are scaled linearly if `fadein`, and the last samples from `t-sec` til `t` are scaled linearly if `fadeout` is enabled.

chaudio.plugins.filters

filters to remove frequency ranges, pass zones, bands, etc

Classes

Butter(**kwargs)	Butterworth filter (https://en.wikipedia.org/wiki/Butterworth_filter), the actuation function based on frequency is nearly linear (in respect to gain in dB), so there not many artifacts around the pass zone

class chaudio.plugins.filters.**Butter**(***kwargs*)

Butterworth filter (https://en.wikipedia.org/wiki/Butterworth_filter), the actuation function based on frequency is nearly linear (in respect to gain in dB), so there not many artifacts around the pass zone

coef(*cutoff*, *hz*, *order*, *btype*)

Internal function for getting the filter coefficients

process(*_data*)

Return the result, with some frequencies filtered out

"order" Butterworth filter order, which should probably stay at `5` (the default)

"cutoff" Frequency, in `hz`, of the cutoff. If `btype` is `highpass`, then anything above `cutoff` remains in the resulting signal (i.e. the high values pass). If `btype=="lowpass"`, all frequencies lower than `cutoff` remain in the signal.

"btype" What filter type? Possible values are "highpass" and "lowpass".

chaudio.plugins.noise

adds white noise to input

Classes

Noise(**kwargs)	adds white noise to input

class chaudio.plugins.noise.**Noise**(***kwargs*)

adds white noise to input

process(*_data*)

Returns the result, with white noise added

"amp" The amplitude of the whitenoise

chaudio.plugins.resolution

changes the minimum resolution

rounds each sample to the nearest value of "step", which has the graphical effect of a "pixelated" waveform (similar to square wave)

Classes

Resolution(**kwargs)	changes the minimum resolution

class chaudio.plugins.resolution.**Resolution**(***kwargs*)

changes the minimum resolution

rounds each sample to the nearest value of "step", which has the graphical effect of a "pixelated" waveform (similar to square wave)

process(*_data*)

Returns the result, but rounded in every step, effectively outputting with less resolution

"norm" If True, then normalize before scaling (default is True), then multiply back so it is effectively untouched. Note you still get the same amplitude back either way, so it isn't permanently normalizing. This is so that the `step` values are treated like proportions (where .5 == 50%), and the result is roughly similar to all inputted waveforms. You should not change this

"step" The value to have the audio data rounded to. If `norm==True`, then this is treated as a proportion. i.e. if `step==.1`, then the resulting audio data only has 19 (`2.0/step-1`) possible amplitudes, which creates interesting effects

chaudio.plugins.volume

A simple multiplier to scale the volume

Classes

Volume(**kwargs)	A simple multiplier to scale the volume.

class chaudio.plugins.volume.**Volume**(***kwargs*)

A simple multiplier to scale the volume. This is effectively the same thing as `kwargs["amp"] * _data`, but being a plugin, it is compatable with other libraries.

process(*_data*)

Returns the result, but amplified

"amp" The amplitude to multiply the source by

Classes

`Basic`(**kwargs)	

3.1.5 chaudio.source

Audio Source (`chaudio.source`)

All these are essentially abstractions above a data array of samples.

Has support for variable number of channels, any samplerate, and data type

Operators are overriden, so that you can apply them to a constant, numpy array, or other audio source

When you set a property (like source.hz or source.channels), the internal data is updated automatically.

Classes

`Mono`(data[, hz])	
`Source`(data[, hz, dtype])	Represents the default audio source, with variable number of channels and samplerate
`Stereo`(data[, hz])	

class `chaudio.source.`**`Source`**(*data*, *hz=None*, *dtype=None*)

Represents the default audio source, with variable number of channels and samplerate

`__init__`(*data*, *hz=None*, *dtype=None*)

Source creation routine

Creates a Source consisting of `data`.

If `data` is a np.ndarray, assume that these are raw sample data, taken at `hz` samplerate (if none is given, `44100`). If no `dtype` is given, default to `data.dtype`

If `data` is a tuple or list, assume that it contains channel data, and set the number of channels to the length of the tuple/list, and each individual channel to the np.ndarray at the corresponding index.

If `data` is *`chaudio.source.Source`*, copy it, but apply the `hz` and `dtype` parameters for the new format. If `hz` isn't given, use `data.hz` as the default, and do the same with `dtype` and `data.dtype`.

If `data` is *`chaudio.arrangers.Arranger`*, calculate its data, and turn into a source.

If `data` is a chaudio class, resample the data to `hz`. Otherwise, assume it is the input was sampled at `hz` per second.

Parameters

- **`data`** (np.ndarray, *`chaudio.source.Source`*, *`chaudio.arrangers.Arranger`*, tuple, list) – Describes how to gather data. See
- **`beats`** (*`int, float`*) – Number of beats (or pulses) per measure
- **`division`** (*`int, float`*) – Note division that represents a single pulse
- **`bpm`** (*`int, optional`*) – The speed, in beats per minute

`copy`()

Returns a copy of the item

Returns An exact copy of the current object

Return type *chaudio.source.Source*

copy_data()
Returns a copy of the raw sample data

Returns Channels with np.ndarray 's describing the sample data

Return type list of np.ndarray

resample(*tohz*)
Internally adjust the sample rate in a smart way (using FFT and IFFT)

This doesn't return anything, so it changes the object it's called on. To return a new source, and not change the one being called, use *chaudio.source.Source.resampled()*

Parameters **tohz** (*int, float*) – The sample rate, in samples per second

resampled(*tohz*)
Returns a copy of the current object resampled to `tohz`

To not make a copy, and instead alter the object in place, use *chaudio.source.Source.resample()*

Parameters **tohz** (*int, float*) – The sample rate, in samples per second

Returns A copy of the object resampled to `tohz` samplerate

Return type *chaudio.source.Source*

rechannel(*tochannels*)
Internally adjust how many channels are stored

This doesn't return anything, so it changes the object it's called on. To return a new source, and not change the one being called, use *chaudio.source.Source.rechanneled()*

If `tochannels == self.channels`, no change is made. Else, the behaviour is thus:

If `tochannels == 1` (which means `self.channels == 2`), the new data array contains 1 item, which is the average of the previous two channels. This will roughly sound the same (as in if you have any sounds that are purely in one channel or the other, they will still be heard).

If `tochannels == 2` (which means `self.channels == 1`), the new data array is the old one, but duplicated.

To return an altered copy, and not change the object itself, use *chaudio.source.Source.rechanneled()*

Parameters **tochannels** (*{ 1, 2 }*) – The number of channels the source should have. Must be `1` or `2`.

rechanneled(*tochannels*)
Return a copy of the object, with a specified number of channels

To not make a copy, and instead alter the object in place, use *chaudio.source.Source.rechannel()*

Parameters **tochannels** (*{ 1, 2 }*) – The number of channels the source should have. Must be `1` or `2`.

Returns A copy of the object with the number of channels changed to `tochannels`

Return type *chaudio.source.Source*

redtype(*todtype*)
Internally adjust what data format is used

This probably shouldn't be used by your application, as it does not rescale values. It's main use is in the *chaudio.util* module, for outputting as WAVE data.

Changes the internal data format

To make a copy, and not alter the current object, use *chaudio.source.Source.redtyped()*

Parameters **todtype** (*{ np.int8, np.int16, np.int32, np.float32, np.float16 }*) – Numpy data format

redtyped(*todtype*)

Return a copy of the object, with a specified internal data format

To not make a copy, and instead alter the object in place, use *chaudio.source.Source.redtype()*

Parameters **todtype** (*{ np.int8, np.int16, np.int32, np.float32, np.float16 }*) – Numpy data format

Returns A copy of the object with the data format changed to `todtype`

Return type *chaudio.source.Source*

__getitem__(*key*)

Return a portion of the data in a source

If `key` is an int or slice, return the channels indicated, in list format. So, use `source[:]` to return all channels as a list, or `source[0]` for the 0th channel (which is left on a stereo source).

If `key` is a tuple, return all the channels represented by `source[key[0]]` subscripted with `key[1]`. So, `source[0, :5]` returns the first 5 samples of the `0``th channel. ``source[:, :5]` returns a list of the first 5 values for each channel.

Parameters **key** (*int, slice, tuple*) – If int or slice, return the channels represented by `key`. If it's a tuple, return channel[key[1]] for each channel represented by `key[0]`. See examples for more info.

Returns If the `key` specified a single channel, return just that channel's specified data as np.ndarray. If multiple channels are indicated, return a list of channel data.

Return type list of np.ndarray or np.ndarray

__setitem__(*key*, *val*)

Set a portion of the data in a source

If `key` is an int or slice, set the channels indicated, in list format. So, use `source[:] = y` to set channels to a y, which must be a list of np.ndarray.

If `key` is a tuple, set all the channels represented by `source[key[0]]` subscripted with `key[1]` to `val`. So, `source[0, :5] = y` sets the first five values of the `0``th channels to y. Note that ``y` must be either a constant, or have the same shape as the values it is replacing. In our example, `y` would have to be a constant, or a np.ndarray with length 5

In general, the following should hold for any source `x`, key `key`, and value `val`:

```
>>> x[key] = val
>>> print (x[key] == val)
True
```

Parameters

- **key** (*int, slice, tuple*) – If int or slice, set the channel data represented by `key`. If it's a tuple, set channel[key[1]] for each channel represented by `key[0]`. See examples for more info.
- **val** (*int, float, list, tuple, np.ndarray*) – The value to set the specified samples to. If it is a list, tuple, or np.ndarray, it must be the same shape as the values it is replacing. So, if saying `x[0, :5] = y`, `y` must be int, float, or `len(y)` must be 5.

Returns If the `key` specified a single channel, return just that channel's specified data as np.ndarray. If multiple channels are indicated, return a list of channel data.

Return type list of np.ndarray or np.ndarray

clear()
Empties all data

To return a copy and not modify the original object, use *chaudio.source.Source.cleared()*.

This empties all data out

cleared()
Empties all data, and returns a copy

To modify in place, *chaudio.source.Source.clear()*.

This empties all data out

insert(*offset*, *_val*)
Inserts samples at a given offset

To return a copy and not modify the original object, use *chaudio.source.Source.inserted()*.

This clears `data[offset:offset+len(_val)]`, and sets it to `_val`

Parameters

- **offset** (*int*) – What sample to insert at
- **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

inserted(*offset*, *_val*)
Returns a copy with inserted samples at a given offset

To not make a copy, and rather edit inplace, use *chaudio.source.Source.insert()*.

Parameters

- **offset** (*int*) – What sample to insert at
- **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

Returns A copy of the object with the `data[offset:offset+len(_val)]` assigned to `_val`.

Return type *chaudio.source.Source*

prepend(*_val*)
Prepend values to the data array

To return a copy and not modify the original object, use *chaudio.source.Source.prepended()*.

This sets `self.data` to `_val` and `data` concatenated. This essentially can be used to add delays, silence, or prepend any other data

Parameters **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

prepended(*_val*)

Returns a copy with prepended values to the data array

To modify the original object and not make a copy, use *chaudio.source.Source.prepend()*.

This returns a copy of the object called on, with `_val` prepended before it.

Parameters **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

Returns `_val` and the object called with concatenated

Return type *chaudio.source.Source*

append(*_val*)

Append values to the data array

To return a copy and not modify the original object, use *chaudio.source.Source.appended()*.

This sets `self.data` to `data` and `_val` concatenated. This tacks on `_val` to the end.

Parameters **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

appended(*_val*)

Returns a copy of the object with values appended to the data array

To not make a copy, and instead modify the object called with, use *chaudio.source.Source.append()*.

Parameters **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

Returns A copy of the current object, but with `_val` appended.

Return type *chaudio.source.Source*

ensure(*length=None*)

Makes sure that the source is a certain length, which will append 0's to the end if needed

To return a copy and not modify the original object, use *chaudio.source.Source.ensured()*.

Parameters **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

ensured(*length=None*)

Makes a copy that is guaranteed to be a certain length, which will append 0's to the end if needed

To modify the original object, use *chaudio.source.Source.ensure()*.

Parameters **_val** (numpy.ndarray, *chaudio.source.Source*, *chaudio.arrangers.Arranger*) – This is converted to a source internally, see *chaudio.source.Source.__init__()* for details on how this is done.

Returns A copy of the object being called, but it is guaranteed to be of a certain length

Return type *chaudio.source.Source*

__weakref__
list of weak references to the object (if defined)

3.1.6 chaudio.util

Utility Functions (chaudio.util)

This module provides useful utilities and classes to be used elsewhere. Note that most of these functions are aliased directly to the main `chaudio` module. Ones that are not aliased are often lower level and may not support all input types.

Functions

cents(hz)	Returns the number of cents (off of 1hz)
concatenate(data)	
ensure_lr_dict(val)	Ensures the input is returned as a dictionary object with right and left specifiers
fft_phase(fft_domain)	
glissando_freq(sfreq, efreq, t[, discrete])	Returns an array of frequencies of a glissando starting at sfreq and ending at efreq
hz(cents)	
lambda_mask(data, qualifier)	
map_domain(domain, chunk, conversion_lambda)	
normalize(v)	Return a scaled version of `v` in the [-1.0, +1.0] range
normalize_factor(v)	The factor needed to scale `v` in order to normalize to [-1.0, +1.0] range
note(name)	Frequency (in hz) of the note as indicated by `name`
sumdup(key, val)	sums duplicates
times(t[, hz])	Returns time sample values
transpose(hz, val[, use_cents])	Transposes a frequency value by a number of cents or semitones

Classes

Chunker(audio[, n, hop])	
FFTChunker(audio[, n, hop])	
TimeSignature(beats, division[, bpm])	Represents a time signature.

chaudio.util.**times**(*t*, *hz=None*)
Returns time sample values

Returns an np.ndarray of time values representing points taken for `t` seconds, at samplerate `hz`.

The length of the resulting object is `t * hz`

Parameters

- **t** (*float, int, np.ndarray, chaudio.source.Source*) – If float or int, it

represents the number of seconds to generate time sample values for. If a numpy ndarray, it assumes the number of seconds is `len(t)/hz`. If it is a chaudio.source.Source, it gets the number of seconds and sample rate (if `hz` is `None`), and uses those.

- **hz** (*float, int*) – The sample rate (samples per second)

Returns An array of time sample values, taken at `hz` samples per second, and lasting `t` (or value derived from `t`) seconds.

Return type np.ndarray

`chaudio.util.`**sumdup**(*key*, *val*)
sums duplicates

`chaudio.util.`**transpose**(*hz*, *val*, *use_cents=True*)
Transposes a frequency value by a number of cents or semitones

Note that if both `hz` and `val` are np arrays, their shapes must be equivalent.

When, `use_cents==True` The effects are thus: +1200 val results in a shift up one octave, -1200 is a shift down one octave.

When, `use_cents==False` The effects are thus: +12 val results in a shift up one octave, -12 is a shift down one octave.

Parameters

- **hz** (*float, int, np.ndarray*) – Frequency, in oscillations per second
- **val** (*float, int, np.ndarray*) – The number of cents (or semitones if `use_cents==False`) to transpose `hz`. It can be positive or negative.
- **use_cents=True** (*bool*) – Whether or not use use cents or semitones

Returns Frequency, in hz, of `hz` shifted by `val`

Return type float

`chaudio.util.`**cents**(*hz*)
Returns the number of cents (off of 1hz)

Parameters **hz** (*float, int, np.ndarray*) – Frequency, in oscillations per second

Returns Cents off of 1 hz

Return type float, np.ndarray

`chaudio.util.`**note**(*name*)
Frequency (in hz) of the note as indicated by `name`

`name` should begin with a note name (like `A`, `B`, `C`, ... , `G`), then optionally a `#` or `b` reflecting a sharp or flat (respectively) tone, and finally an optional octave number (starting with `0` up to `8`).

If no octave number is given, it defaults to `4`.

Parameters **name** (*str*) – String representation of a note, like `A` or `C#5`

Returns Frequency, in hz, of the note described by `name`

Return type float

Examples

```
>>> chaudio.note("A")
440.0
>>> chaudio.note("A5")
880.0
>>> chaudio.note("A#5")
932.327523
>>> chaudio.note("TESTING7")
ValueError: invalid note name: TESTING
```

chaudio.util.**ensure_lr_dict**(*val*)

Ensures the input is returned as a dictionary object with right and left specifiers

If `val` is a dictionary, look for `left` or `right` keys. If both exist, return those as a new dictionary. If only one exists, assume that value stands for both sides.

If `val` is a tuple/list, and it has 1 value, assume that is for both left and right. If it has length of 2, assum `val[0]` is the left value and `val[1]` is right.

Else, assume the single `val` is the value for left and right, i.e. there is no difference between the two sides.

If `val` does not fit these rules, a `ValueException` is raised.

Parameters **val** (*any*) – value to be ensured as a left/right dictionary

Returns Value with keys 'left' and 'right', determined by the input value

Return type dict

chaudio.util.**normalize_factor**(*v*)

The factor needed to scale `v` in order to normalize to [-1.0, +1.0] range

In the case that `v` is a chaudio.source.Source, return the highest of any sample in any channel.

Parameters **v** (`chaudio.source.Source` *or* *np.ndarray*) – The collection of amplitudes

Returns The highest maximum amplitude of the absolute value of `v`

Return type float

chaudio.util.**normalize**(*v*)

Return a scaled version of `v` in the [-1.0, +1.0] range

Normalize `v` such that all values are scaled by the same linear factor, and $-1.0 <= k <= +1.0$ for all values `k` in `v`. If given a chaudio.source.Source, all channels are scaled by the same factor (so that the channels will still be even).

Parameters **v** (`chaudio.source.Source` *or* *np.ndarray*) – The source being normalized

Returns `v` such that all amplitudes have been scaled to fit inside the [-1.0, +1.0] range

Return type *chaudio.source.Source* or np.ndarray

class chaudio.util.**TimeSignature**(*beats*, *division*, *bpm=60*)

Represents a time signature.

__init__(*beats*, *division*, *bpm=60*)

TimeSignature creation routine

Return a time signature representing measures each with `beats` beats (or pulses).

The note represented as `division` getting a single beat.

If division is 4, the quarter note gets the beat, 8 means the 8th note gets the beat, and so on.

Parameters

- **v** (*chaudio.source.Source or np.ndarray*) – The source being normalized
- **beats** (*int, float*) – Number of beats (or pulses) per measure
- **division** (*int, float*) – Note division that represents a single pulse
- **bpm** (*int, optional*) – The speed, in beats per minute

__getitem__(*key*)

Returns the time in seconds of a number of beats, or a number of measures and beats.

(this method is an alias for subscripting, so `tsig.__getitem__(key)` is equivelant to `tsig[key]`)

If `key` is a tuple, return the number of seconds that is equivalant to `key[0]` measures, and `key[1]` beats.

In all cases, `tsig[a] == tsig[a//tsig.beats, a%tsig.beats]`.

When calling using a `key` that is a tuple, `key[1]` must not exceed the number of beats. This is to prevent errors arising from improper lengths. However, the number of beats can be any non-negative value if using a key that is a float or int.

Parameters **key** (*int, float, tuple*) – Either a tuple containing (measure, beat), or a number of beats

Returns The amount of time that `key` represents (in seconds)

Return type float

__weakref__

list of weak references to the object (if defined)

chaudio.util.**glissando_freq**(*sfreq*, *efreq*, *t*, *discrete=False*)

Returns an array of frequencies of a glissando starting at sfreq and ending at efreq

Parameters

- **sfreq** (*float*) – Frequency at the start of the time array
- **efreq** (*float*) – Frequency at the end of the time array
- **t** (*np.ndarray*) – Array of time sample values
- **discrete** (*bool*) – Default=False, but if true, they are truncated to note values

Returns Array of frequencies in corresponding to *t*

Return type np.ndarray

3.1.7 chaudio.waves

Waveform Generation Functions (chaudio.waves)

Source for calculating waveform values (like *chaudio.waves.sin()*, *chaudio.waves.saw()*, etc)

In general, all waveform functions `f` should take in a time parameter `t` that can be either a constant or a numpy array, and *hz* should be able to be a constant or numpy array. Also, they should accept an optional value called `tweak`, which (if supported) should return a slightly different waveform based on the value of `tweak`.

Note that ALL waveforms should accept this `tweak` value, even if they do nothing. This is for compatability

Some other general rules (these are by no means required, however):

```
>>> f(0, hz) == 0
>>> f(t, hz, tweak=None) == f(t, hz)
>>> f(t+1.0/hz, hz) == f(t, hz)
```

The general idea with a waveform is that it repeats every `1.0/hz` seconds, and each oscillation (or cycle) is the exact same.

Different frequencies have different pitches (see *chaudio.util.note()*), and different waveforms have different timbres (pronounced `TAM-BER` or `TIM-BER`). In fact, all instruments digital and real-world all are just different waveforms. Even when you play a guitar, it is simply a waveform played at a pitch.

These are generated on a non-continious (which is a synonym of discrete) sample array, each value in the sample array representing a point in time, and the amplitude of the sound at that point in time. These sample arrays, since they aren't continious, have a samplerate, or how many records it has per second. The most common value is 44100, that is, 44100 recordings are held per second of data. And, consequently, if our array is named `ar`, `ar[0]` represents the sound's amplitude at time $t = \frac{0}{44100} = 0$‘ seconds. And, at `ar[25000]`, it holds the amplitude at $t = \frac{25000}{44100} \approx .566$ seconds.

As we have said, the waveform repeats every $\frac{1}{hz}$‘ seconds, which means that it repeats hz times per second. Thus, $ar[0]$ represents the start of the first waveform, and $ar[\lfloor \frac{44100}{hz} \rfloor]$ marks the end of the first oscillation and the beginning of the second.

Functions

`noise`(t[, hz, tweak])	
phase_correction(t, hz)	Computes the phase correction factor for time values and frequencies
saw(t, hz[, tweak])	Computes the sawtooth wave of sample times (`t`), and frequencies (`hz`)
sin(t, hz[, tweak])	Computes the sin wave of sample times (`t`), and frequencies (`hz`)
square(t, hz[, tweak])	Computes the square wave of sample times (`t`), and frequencies (`hz`)
triangle(t, hz[, tweak])	Computes the triangle wave of sample times (`t`), and frequencies (`hz`)
`zero`(t[, hz, tweak])	

`chaudio.waves.`**`phase_correction`**(*t*, *hz*)

Computes the phase correction factor for time values and frequencies

See this post for more info

Parameters

- **`t`** (*np.ndarray*) – The time sample values
- **`hz`** (*float, int, np.ndarray*) – Frequency of wave. If the type of `hz` is np.ndarray, it must have the same shape as `t`, and in that case each corresponding value of `t`'s wave is given a phase correction value

Returns Returns the phase correction vector needed to correctly produce sound with a changing frequency

Return type np.ndarray

chaudio.waves.**sin**(*t*, *hz*, *tweak=None*)

Computes the sin wave of sample times (t), and frequencies (hz)

Optionally, if tweak is set, return a slightly modified waveform.

With no tweak, the return value is $sin(2\pi * hz * t)$, but with the return value, $sin(2\pi * hz * t)^{1+tweak}$ is returned.

Parameters

- **t** (*float, int, np.ndarray*) – If a float or int, return the value of the sin wave at time t, in seconds. If it is a numpy array, return an array of values at the sin wave corresponding to each time value in the array.
- **hz** (*float, int, np.ndarray*) – Frequency of wave. If the type of hz is np.ndarray, it must have the same shape as t, and in that case each corresponding value of t's wave is assumed to have hz's value at the same index as the frequency value.
- **tweak** (*float, int, np.ndarray*) – A value to change the waveform. If the type of tweak is a numpy array, it must have the same shape as t, and in that case each corresponding value of t's wave is assumed to have tweak's value at the same index as the tweak value (see examples below).

Returns If all t, hz, and tweak are floats or ints, the function returns a float. Else, all parameters which are np.ndarray's must have the same shape, and the returned value is the same shape.

Return type float, np.ndarray

Examples

```
>>> t = 0
>>> chaudio.waves.sin(t, 1)
0.0
>>> t = chaudio.times(5)
>>> chaudio.waves.sin(t, 1)
array([ 0.        ,  0.00014248,  0.00028495, ..., -0.00042743,
   -0.00028495, -0.00014248])
```

See also:

chaudio.util.times() returns sample times, which can be passed to this function as sample times

chaudio.waves.**saw**(*t*, *hz*, *tweak=None*)

Computes the sawtooth wave of sample times (t), and frequencies (hz)

Optionally, if tweak is set, return a slightly modified waveform.

With no tweak, the return value is $saw(2\pi * hz * t)$, but with the return value, $saw(2\pi * hz * t) * (1 + tweak * sin(t, hz, tweak))$ is returned.

This has the effect of making the waveform appear "bendy", but still resemble a sawtooth.

Parameters

- **t** (*float, int, np.ndarray*) – If a float or int, return the value of the sawtooth wave at time t, in seconds. If it is a numpy array, return an array of values at the sawtooth wave corresponding to each time value in the array.
- **hz** (*float, int, np.ndarray*) – Frequency of wave. If the type of hz is np.ndarray, it must have the same shape as t, and in that case each corresponding value of t's wave is assumed to have hz's value at the same index as the frequency value.

- **tweak** (*float, int, np.ndarray*) – A value to change the waveform. If the type of `tweak` is a numpy array, it must have the same shape as `t`, and in that case each corresponding value of `t`'s wave is assumed to have `tweak`'s value at the same index as the tweak value (see examples below).

Returns If all `t`, `hz`, and `tweak` are floats or ints, the function returns a float. Else, all parameters which are np.ndarray's must have the same shape, and the returned value is the same shape.

Return type float, np.ndarray

Examples

```
>>> t = 0
>>> chaudio.waves.saw(t, 1)
0.0
>>> t = chaudio.times(5)
>>> chaudio.waves.saw(t, 1)
array([  0.00000000e+00,   4.53514739e-05,   9.07029478e-05, ...,
    -1.36054422e-04,  -9.07029478e-05,  -4.53514739e-05])
```

See also:

chaudio.util.times() returns sample times, which can be passed to this function as sample times

chaudio.waves.**square**(*t*, *hz*, *tweak=None*)

Computes the square wave of sample times (`t`), and frequencies (`hz`)

Optionally, if tweak is set, return a slightly modified waveform.

With no tweak, the return value is $square(2\pi * hz * t)$, which has a duty cycle of `50%`, or `.5`. If set, the duty cycle is set to `tweak`, and if `tweak==.5`, that results in a normal square wave.

This has similar effects to opening up an envelope

Parameters

- **t** (*float, int, np.ndarray*) – If a float or int, return the value of the square wave at time `t`, in seconds. If it is a numpy array, return an array of values at the square wave corresponding to each time value in the array.
- **hz** (*float, int, np.ndarray*) – Frequency of wave. If the type of `hz` is np.ndarray, it must have the same shape as `t`, and in that case each corresponding value of `t`'s wave is assumed to have `hz`'s value at the same index as the frequency value.
- **tweak** (*float, int, np.ndarray*) – A value to change the waveform. If the type of `tweak` is a numpy array, it must have the same shape as `t`, and in that case each corresponding value of `t`'s wave is assumed to have `tweak`'s value at the same index as the tweak value (see examples below).

Returns If all `t`, `hz`, and `tweak` are floats or ints, the function returns a float. Else, all parameters which are np.ndarray's must have the same shape, and the returned value is the same shape.

Return type float, np.ndarray

Notes

Unlike most other waveforms, the square wave starts at `-1`, whereas most start at `0`. However, since the square wave only has values taking either `-1` or `+1` (even in modified form), this is done as a compromise

Examples

```
>>> t = 0
>>> chaudio.waves.square(t, 1)
-1
>>> t = chaudio.times(5)
>>> chaudio.waves.saw(t, 1)
array([-1, -1, -1, ...,  1,  1,  1])
```

See also:

chaudio.util.times() : returns sample times, which can be passed to this function as sample times Pulse wave : with a modified tweak value, the waveform is a Pulse wave with duty cycle equal to `tweak`

chaudio.waves.**triangle**(*t*, *hz*, *tweak=None*)

Computes the triangle wave of sample times (`t`), and frequencies (`hz`)

Optionally, if tweak is set, return a slightly modified waveform.

With no tweak, the return value is $triangle(2\pi * hz * t)$, which looks like a sin wave, except it is straight lines.

With a tweak value, $triangle(2\pi * hz * t) - tweak * saw(2\pi * hz * t) * square(2\pi * hz * t, tweak)$ is returned, which can generate a lot of different timbres. In the future, I'll add an in depth description of what kind of sounds this creates.

Parameters

- **t** (*float, int, np.ndarray*) – If a float or int, return the value of the triangle wave at time `t`, in seconds. If it is a numpy array, return an array of values at the triangle wave corresponding to each time value in the array.
- **hz** (*float, int, np.ndarray*) – Frequency of wave. If the type of `hz` is np.ndarray, it must have the same shape as `t`, and in that case each corresponding value of `t`'s wave is assumed to have `hz`'s value at the same index as the frequency value.
- **tweak** (*float, int, np.ndarray*) – A value to change the waveform. If the type of `tweak` is a numpy array, it must have the same shape as `t`, and in that case each corresponding value of `t`'s wave is assumed to have `tweak`'s value at the same index as the tweak value (see examples below).

Returns If all `t`, `hz`, and `tweak` are floats or ints, the function returns a float. Else, all parameters which are np.ndarray's must have the same shape, and the returned value is the same shape.

Return type float, np.ndarray

Notes

Unlike most other waveforms, the square wave starts at `-1`, whereas most start at `0`. However, since the square wave only has values taking either `-1` or `+1` (even in modified form), this is done as a compromise

Examples

```
>>> t = 0
>>> chaudio.waves.triangle(t, 1)
0.0
>>> t = chaudio.times(5)
>>> chaudio.waves.triangle(t, 1)
```

```
array([  0.00000000e+00,   9.07029478e-05,   1.81405896e-04, ...,
    -2.72108844e-04,  -1.81405896e-04,  -9.07029478e-05])
```

See also:

chaudio.util.times() returns sample times, which can be passed to this function as sample times

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

chaudio.arrangers, 7
chaudio.instruments, 10
chaudio.io, 15
chaudio.plugins, 17
chaudio.plugins.echo, 18
chaudio.plugins.fade, 18
chaudio.plugins.filters, 19
chaudio.plugins.noise, 19
chaudio.plugins.resolution, 20
chaudio.plugins.volume, 20
chaudio.source, 21
chaudio.util, 26
chaudio.waves, 29

Index

Symbols

__getitem__() (chaudio.instruments.Instrument method), 12
__getitem__() (chaudio.source.Source method), 23
__getitem__() (chaudio.util.TimeSignature method), 29
__init__() (chaudio.arrangers.Arranger method), 8
__init__() (chaudio.arrangers.ExtendedArranger method), 9
__init__() (chaudio.arrangers.InsertCall method), 8
__init__() (chaudio.instruments.ADSREnvelope method), 13
__init__() (chaudio.instruments.Instrument method), 11
__init__() (chaudio.instruments.MultiOscillator method), 15
__init__() (chaudio.instruments.Oscillator method), 14
__init__() (chaudio.source.Source method), 21
__init__() (chaudio.util.TimeSignature method), 28
__setitem__() (chaudio.instruments.Instrument method), 12
__setitem__() (chaudio.source.Source method), 23
__weakref__ (chaudio.arrangers.Arranger attribute), 9
__weakref__ (chaudio.arrangers.InsertCall attribute), 8
__weakref__ (chaudio.instruments.ADSREnvelope attribute), 14
__weakref__ (chaudio.instruments.Instrument attribute), 12
__weakref__ (chaudio.source.Source attribute), 26
__weakref__ (chaudio.util.TimeSignature attribute), 29

A

add_final_plugin() (chaudio.arrangers.Arranger method), 9
add_insert_plugin() (chaudio.arrangers.Arranger method), 9
add_plugin() (chaudio.instruments.Instrument method), 11
ADSREnvelope (class in chaudio.instruments), 12
append() (chaudio.source.Source method), 25
appended() (chaudio.source.Source method), 25
apply_insert() (chaudio.arrangers.Arranger method), 9
Arranger (class in chaudio.arrangers), 8

B

Butter (class in chaudio.plugins.filters), 19

C

calc_val() (chaudio.instruments.ADSREnvelope method), 13
cents() (in module chaudio.util), 27
chaudio (module), 7
chaudio.arrangers (module), 7
chaudio.instruments (module), 10
chaudio.io (module), 15
chaudio.plugins (module), 17
chaudio.plugins.echo (module), 18
chaudio.plugins.fade (module), 18
chaudio.plugins.filters (module), 19
chaudio.plugins.noise (module), 19
chaudio.plugins.resolution (module), 20
chaudio.plugins.volume (module), 20
chaudio.source (module), 21
chaudio.util (module), 26
chaudio.waves (module), 29
clear() (chaudio.source.Source method), 24
cleared() (chaudio.source.Source method), 24
coef() (chaudio.plugins.filters.Butter method), 19
copy() (chaudio.instruments.Instrument method), 12
copy() (chaudio.source.Source method), 21
copy_data() (chaudio.source.Source method), 22

E

Echo (class in chaudio.plugins.echo), 18
ensure() (chaudio.source.Source method), 25
ensure_lr_dict() (in module chaudio.util), 28
ensured() (chaudio.source.Source method), 25
ExtendedArranger (class in chaudio.arrangers), 9

F

Fade (class in chaudio.plugins.fade), 18

fromfile() (in module chaudio.io), 16
fromstring() (in module chaudio.io), 17

G

getarg() (chaudio.arrangers.Arranger method), 8
glissando_freq() (in module chaudio.util), 29

I

insert() (chaudio.source.Source method), 24
insert_beat() (chaudio.arrangers.ExtendedArranger method), 10
insert_sample() (chaudio.arrangers.Arranger method), 9
insert_time() (chaudio.arrangers.ExtendedArranger method), 10
InsertCall (class in chaudio.arrangers), 8
inserted() (chaudio.source.Source method), 24
Instrument (class in chaudio.instruments), 11

M

merged_kwargs() (chaudio.instruments.Instrument method), 12
MultiOscillator (class in chaudio.instruments), 15

N

Noise (class in chaudio.plugins.noise), 19
normalize() (in module chaudio.util), 28
normalize_factor() (in module chaudio.util), 28
note() (chaudio.instruments.Instrument method), 11
note() (in module chaudio.util), 27

O

Oscillator (class in chaudio.instruments), 14

P

phase_correction() (in module chaudio.waves), 30
play() (in module chaudio.io), 16
prepend() (chaudio.source.Source method), 24
prepended() (chaudio.source.Source method), 25
process() (chaudio.plugins.echo.Echo method), 18
process() (chaudio.plugins.fade.Fade method), 18
process() (chaudio.plugins.filters.Butter method), 19
process() (chaudio.plugins.noise.Noise method), 19
process() (chaudio.plugins.resolution.Resolution method), 20
process() (chaudio.plugins.volume.Volume method), 20

R

raw_note() (chaudio.instruments.Instrument method), 11
raw_note() (chaudio.instruments.Oscillator method), 15
rechannel() (chaudio.source.Source method), 22
rechanneled() (chaudio.source.Source method), 22
redtype() (chaudio.source.Source method), 22
redtyped() (chaudio.source.Source method), 23
remove_plugin() (chaudio.instruments.Instrument method), 11
resample() (chaudio.source.Source method), 22
resampled() (chaudio.source.Source method), 22
Resolution (class in chaudio.plugins.resolution), 20

S

saw() (in module chaudio.waves), 31
setarg() (chaudio.arrangers.Arranger method), 8
sin() (in module chaudio.waves), 30
Source (class in chaudio.source), 21
square() (in module chaudio.waves), 32
sumdup() (in module chaudio.util), 27

T

times() (in module chaudio.util), 26
TimeSignature (class in chaudio.util), 28
tofile() (in module chaudio.io), 17
tostring() (in module chaudio.io), 17
transpose() (in module chaudio.util), 27
triangle() (in module chaudio.waves), 33

V

Volume (class in chaudio.plugins.volume), 20