auditok Documentation

Release v0.2.0

Amine Sehili

Mar 02, 2021

Getting started

1 Installation 3

2 Load audio data 5
2.1 From a file . . . 5
2.2 From a *bytes* object . . . 5
2.3 From the microphone . . . 6
2.4 Skip part of audio data . . . 6
2.5 Limit the amount of read audio . . . 6

3 Basic split example 7

4 Split and plot 9

5 Read and split data from the microphone 11

6 Access recorded data after split 13

7 Working with AudioRegions 15
7.1 Basic region information . . . 15
7.2 Concatenate regions . . . 15
7.3 Repeat a region . . . 16
7.4 Split one region into N regions of equal size . . . 16
7.5 Slice a region by samples, seconds or milliseconds . . . 16
7.6 Get arrays of audio samples . . . 17

8 Read and split audio data online 19

9 Read audio data with an external program 21

10 Play back audio detections 23

11 Print out detection information 25

12 Save detections 27

13 Save whole audio stream 29

14 Plot detections 31

15 Core 33
15.1 auditok.core.load 33
15.2 auditok.core.split 34
15.3 auditok.core.AudioRegion 36
15.4 auditok.core.StreamTokenizer 37

16 Util 47
16.1 auditok.util.AudioEnergyValidator 47
16.2 auditok.util.AudioReader 48
16.3 auditok.util.Recorder 50
16.4 auditok.util.make_duration_formatter 50
16.5 auditok.util.make_channel_selector 51

17 Low-level IO 59
17.1 auditok.io.AudioSource 59
17.2 auditok.io.Rewindable 60
17.3 auditok.io.BufferAudioSource 61
17.4 auditok.io.WaveAudioSource 62
17.5 auditok.io.PyAudioSource 62
17.6 auditok.io.StdinAudioSource 63
17.7 auditok.io.PyAudioPlayer 64
17.8 auditok.io.from_file 64
17.9 auditok.io.to_file 65
17.10 auditok.io.player_for 66

18 Signal processing 73
18.1 auditok.signal.to_array 73
18.2 auditok.signal.extract_single_channel 74
18.3 auditok.signal.compute_average_channel 74
18.4 auditok.signal.compute_average_channel_stereo 74
18.5 auditok.signal.separate_channels 74
18.6 auditok.signal.calculate_energy_single_channel 75
18.7 auditok.signal.calculate_energy_multichannel 75

19 Dataset 79
19.1 auditok.dataset.one_to_six_arabic_16000_mono_bc_noise 79
19.2 auditok.dataset.was_der_mensch_saet_mono_44100_lead_trail_silence 79

20 License 81

Python Module Index 83

Index 85

`auditok` is an **Audio Activity Detection** tool that can process online data (read from an audio device or from standard input) as well as audio files. It can be used as a command line program or by calling its API.

CHAPTER 1

Installation

A basic version of `auditok` will run with standard Python (>=3.4). However, without installing additional dependencies, `auditok` can only deal with audio files in *wav* or *raw* formats. if you want more features, the following packages are needed:

- pydub : read audio files in popular audio formats (ogg, mp3, etc.) or extract audio from a video file.
- pyaudio : read audio data from the microphone and play audio back.
- tqdm : show progress bar while playing audio clips.
- matplotlib : plot audio signal and detections.
- numpy : required by matplotlib. Also used for some math operations instead of standard python if available.

Install the latest stable version with pip:

```
sudo pip install auditok
```

Install with the latest development version from github:

```
pip install git+https://github.com/amsehili/auditok
```

or

```
git clone https://github.com/amsehili/auditok.git
cd auditok
python setup.py install
```

CHAPTER 2

Load audio data

Audio data is loaded with the `load()` function which can read from audio files, the microphone or use raw audio data.

2.1 From a file

If the first argument of `load()` is a string, it should be a path to an audio file.

```
import auditok
region = auditok.load("audio.ogg")
```

If input file contains raw (headerless) audio data, passing *audio_format="raw"* and other audio parameters (*sampling_rate*, *sample_width* and *channels*) is mandatory. In the following example we pass audio parameters with their short names:

```
region = auditok.load("audio.dat",
                      audio_format="raw",
                      sr=44100, # alias for `sampling_rate`
                      sw=2      # alias for `sample_width`
                      ch=1      # alias for `channels`
                      )
```

2.2 From a *bytes* object

If the type of the first argument *bytes*, it's interpreted as raw audio data:

```
sr = 16000
sw = 2
ch = 1
data = b"\0" * sr * sw * ch
```

(continues on next page)

(continued from previous page)

```
region = auditok.load(data, sr=sr, sw=sw, ch=ch)
print(region)
# alternatively you can use
#region = auditok.AudioRegion(data, sr, sw, ch)
```

output:

```
AudioRegion(duration=1.000, sampling_rate=16000, sample_width=2, channels=1)
```

2.3 From the microphone

If the first argument is *None*, `load()` will try to read data from the microphone. Audio parameters, as well as the *max_read* parameter are mandatory:

```
sr = 16000
sw = 2
ch = 1
five_sec_audio = load(None, sr=sr, sw=sw, ch=ch, max_read=5)
print(five_sec_audio)
```

output:

```
AudioRegion(duration=5.000, sampling_rate=16000, sample_width=2, channels=1)
```

2.4 Skip part of audio data

If the *skip* parameter is > 0, `load()` will skip that amount in seconds of leading audio data:

```
import auditok
region = auditok.load("audio.ogg", skip=2) # skip the first 2 seconds
```

This argument must be 0 when reading data from the microphone.

2.5 Limit the amount of read audio

If the *max_read* parameter is > 0, `load()` will read at most that amount in seconds of audio data:

```
import auditok
region = auditok.load("audio.ogg", max_read=5)
assert region.duration <= 5
```

This argument is mandatory when reading data from the microphone.

CHAPTER 3

Basic split example

In the following we'll use the `split()` function to tokenize an audio file, requiring that valid audio events be at least 0.2 second long, at most 4 seconds long and contain a maximum of 0.3 second of continuous silence. Limiting the size of detected events to 4 seconds means that an event of, say, 9.5 seconds will be returned as two 4-second events plus a third 1.5-second event. Moreover, a valid event might contain many *silences* as far as none of them exceeds 0.3 second.

`split()` returns a generator of `AudioRegion`. An `AudioRegion` can be played, saved, repeated (i.e., multiplied by an integer) and concatenated with another region (see examples below). Notice that `AudioRegion` objects returned by `split()` have a `start` a `stop` information stored in their meta data that can be accessed like *object.meta.start*.

```python
import auditok

# split returns a generator of AudioRegion objects
audio_regions = auditok.split(
    "audio.wav",
    min_dur=0.2,      # minimum duration of a valid audio event in seconds
    max_dur=4,        # maximum duration of an event
    max_silence=0.3, # maximum duration of tolerated continuous silence within an␣
↪event
    energy_threshold=55 # threshold of detection
)

for i, r in enumerate(audio_regions):

    # Regions returned by `split` have 'start' and 'end' metadata fields
    print("Region {i}: {r.meta.start:.3f}s -- {r.meta.end:.3f}s".format(i=i, r=r))

    # play detection
    # r.play(progress_bar=True)

    # region's metadata can also be used with the `save` method
    # (no need to explicitly specify region's object and `format` arguments)
    filename = r.save("region_{meta.start:.3f}-{meta.end:.3f}.wav")
```

(continues on next page)

(continued from previous page)

```
    print("region saved as: {}".format(filename))
```

output example:

```
Region 0: 0.700s -- 1.400s
region saved as: region_0.700-1.400.wav
Region 1: 3.800s -- 4.500s
region saved as: region_3.800-4.500.wav
Region 2: 8.750s -- 9.950s
region saved as: region_8.750-9.950.wav
Region 3: 11.700s -- 12.400s
region saved as: region_11.700-12.400.wav
Region 4: 15.050s -- 15.850s
region saved as: region_15.050-15.850.wav
```

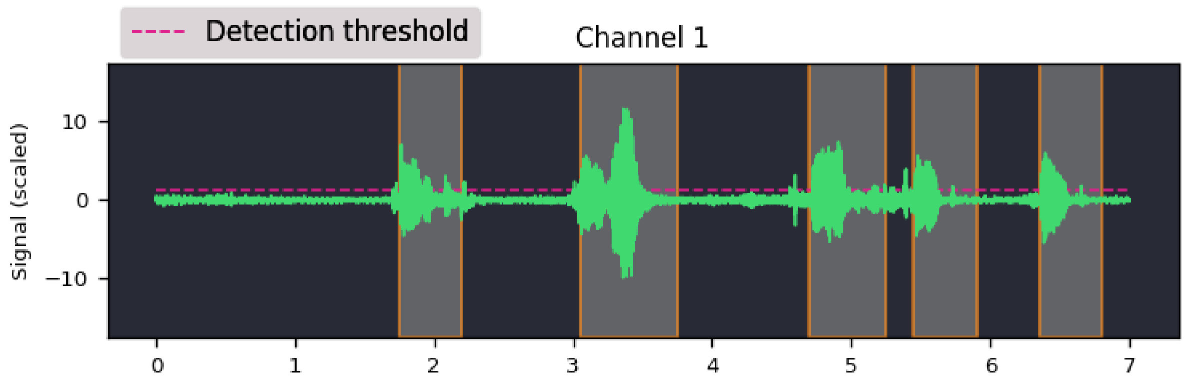
CHAPTER 4

Split and plot

Visualize audio signal and detections:

```
import auditok
region = auditok.load("audio.wav") # returns an AudioRegion object
regions = region.split_and_plot(...) # or just region.splitp()
```

output figure:



CHAPTER 5

Read and split data from the microphone

If the first argument of `split()` is None, audio data is read from the microphone (requires pyaudio):

```
import auditok

sr = 16000
sw = 2
ch = 1
eth = 55 # alias for energy_threshold, default value is 50

try:
    for region in auditok.split(input=None, sr=sr, sw=sw, ch=ch, eth=eth):
        print(region)
        region.play(progress_bar=True) # progress bar requires `tqdm`
except KeyboardInterrupt:
     pass
```

`split()` will continue reading audio data until you press `Ctrl-C`. If you want to read a specific amount of audio data, pass the desired number of seconds with the *max_read* argument.

CHAPTER 6

Access recorded data after split

Using a Recorder object you can get hold of acquired audio data:

```
import auditok

sr = 16000
sw = 2
ch = 1
eth = 55 # alias for energy_threshold, default value is 50

rec = auditok.Recorder(input=None, sr=sr, sw=sw, ch=ch)

try:
    for region in auditok.split(rec, sr=sr, sw=sw, ch=ch, eth=eth):
        print(region)
        region.play(progress_bar=True) # progress bar requires `tqdm`
except KeyboardInterrupt:
     pass

rec.rewind()
full_audio = load(rec.data, sr=sr, sw=sw, ch=ch)
# alternatively you can use
full_audio = auditok.AudioRegion(rec.data, sr, sw, ch)
```

Recorder also accepts a *max_read* argument.

CHAPTER 7

Working with AudioRegions

The following are a couple of interesting operations you can do with `AudioRegion` objects.

7.1 Basic region information

```
import auditok
region = auditok.load("audio.wav")
len(region) # number of audio samples int the regions, one channel considered
region.duration # duration in seconds
region.sampling_rate # alias `sr`
region.sample_width # alias `sw`
region.channels # alias `ch`
```

7.2 Concatenate regions

```
import auditok
region_1 = auditok.load("audio_1.wav")
region_2 = auditok.load("audio_2.wav")
region_3 = region_1 + region_2
```

Particularly useful if you want to join regions returned by `split()`:

```
import auditok
regions = auditok.load("audio.wav").split()
gapless_region = sum(regions)
```

7.3 Repeat a region

Multiply by a positive integer:

```
import auditok
region = auditok.load("audio.wav")
region_x3 = region * 3
```

7.4 Split one region into N regions of equal size

Divide by a positive integer (this has nothing to do with silence-based tokenization):

```
import auditok
region = auditok.load("audio.wav")
regions = regions / 5
assert sum(regions) == region
```

Note that if no perfect division is possible, the last region might be a bit shorter than the previous N-1 regions.

7.5 Slice a region by samples, seconds or milliseconds

Slicing an `AudioRegion` can be interesting in many situations. You can for example remove a fixed-size portion of audio data from the beginning or from the end of a region or crop a region by an arbitrary amount as a data augmentation strategy.

The most accurate way to slice an *AudioRegion* is to use indices that directly refer to raw audio samples. In the following example, assuming that the sampling rate of audio data is 16000, you can extract a 5-second region from main region, starting from the 20th second as follows:

```
import auditok
region = auditok.load("audio.wav")
start = 20 * 16000
stop = 25 * 16000
five_second_region = region[start:stop]
```

This allows you to practically start and stop at any audio sample within the region. Just as with a *list* you can omit one of *start* and *stop*, or both. You can also use negative indices:

```
import auditok
region = auditok.load("audio.wav")
start = -3 * region.sr # `sr` is an alias of `sampling_rate`
three_last_seconds = region[start:]
```

While slicing by raw samples is flexible, slicing with temporal indices is more intuitive. You can do so by accessing the `millis` or `seconds` views of an *AudioRegion* (or their shortcut alias *ms* and *sec* or *s*).

With the `millis` view:

```
import auditok
region = auditok.load("audio.wav")
five_second_region = region.millis[5000:10000]
```

or with the `seconds` view:

```
import auditok
region = auditok.load("audio.wav")
five_second_region = region.seconds[5:10]
```

`seconds` indices can also be floats:

```
import auditok
region = auditok.load("audio.wav")
five_second_region = region.seconds[2.5:7.5]
```

7.6 Get arrays of audio samples

If *numpy* is not installed, the *samples* attributes is a list of audio samples arrays (standard *array.array* objects), one per channels. If numpy is installed, *samples* is a 2-D *numpy.ndarray* where the fist dimension is the channel and the second is the the sample.

```
import auditok
region = auditok.load("audio.wav")
samples = region.samples
assert len(samples) == region.channels
```

If *numpy* is installed you can use:

```
import numpy as np
region = auditok.load("audio.wav")
samples = np.asarray(region)
assert len(samples.shape) == 2
```

`auditok` can also be used from the command-line. For more information about parameters and their description type:

```
auditok -h
```

In the following we'll a few examples that covers most use-cases.

CHAPTER 8

Read and split audio data online

To try `auditok` from the command line with you voice, you should either install pyaudio so that `auditok` can directly read data from the microphone, or record data with an external program (e.g., *sox*) and redirect its output to `auditok`.

Read data from the microphone (*pyaudio* installed):

```
auditok
```

This will print the *id*, *start time* and *end time* of each detected audio event. Note that we didn't pass any additional arguments to the previous command, so `auditok` will use default values. The most important arguments are:

- `-n`, `--min-duration` : minimum duration of a valid audio event in seconds, default: 0.2
- `-m`, `--max-duration` : maximum duration of a valid audio event in seconds, default: 5
- `-s`, `--max-silence` : maximum duration of a consecutive silence within a valid audio event in seconds, default: 0.3
- `-e`, `--energy-threshold` : energy threshold for detection, default: 50

CHAPTER 9

Read audio data with an external program

If you don't have *pyaudio*, you can use *sox* for data acquisition (*sudo apt-get install sox*) and make `auditok` read data from standard input:

```
rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok - -r 16000 -w 2 -c 1
```

Note that when data is read from standard input, the same audio parameters must be used for both *sox* (or any other data generation/acquisition tool) and `auditok`. The following table summarizes audio parameters.

Audio parameter	sox option	*auditok* option	*auditok* default
Sampling rate	-r	-r	16000
Sample width	-b (bits)	-w (bytes)	2
Channels	-c	-c	1
Encoding	-e	NA	always a signed int

According to this table, the previous command can be run with the default parameters as:

```
rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok -i -
```

CHAPTER 10

Play back audio detections

Use the -E option (for echo):

```
auditok -E
# or
rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok - -E
```

The second command works without further argument because data is recorded with `auditok`'s default audio parameters . If one of the parameters is not at the default value you should specify it alongside `-E`.

Using `-E` requires *pyaudio*, if it's not installed you can use the `-C` (used to run an external command with detected audio event as argument):

```
rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok - -C "play -q {file}"
```

Using the `-C` option, `auditok` will save a detected event to a temporary wav file, fill the `{file}` placeholder with the temporary name and run the command. In the above example we used `-C` to play audio data with an external program but you can use it to run any other command.

CHAPTER 11

Print out detection information

By default `auditok` prints out the **id**, the **start** and the **end** of each detected audio event. The latter two values represent the absolute position of the event within input stream (file or microphone) in seconds. The following listing is an example output with the default format:

```
1 1.160 2.390
2 3.420 4.330
3 5.010 5.720
4 7.230 7.800
```

The format of the output is controlled by the `--printf` option. Alongside `{id}`, `{start}` and `{end}` placeholders, you can use `{duration}` and `{timestamp}` (system timestamp of detected event) placeholders.

Using the following format for example:

```
auditok audio.wav  --printf "{id}: [{timestamp}] start:{start}, end:{end}, dur:
↪{duration}"
```

the output would be something like:

```
1: [2021/02/17 20:16:02] start:1.160, end:2.390, dur: 1.230
2: [2021/02/17 20:16:04] start:3.420, end:4.330, dur: 0.910
3: [2021/02/17 20:16:06] start:5.010, end:5.720, dur: 0.710
4: [2021/02/17 20:16:08] start:7.230, end:7.800, dur: 0.570
```

The format of `{timestamp}` is controlled by `--timestamp-format` (default: *"%Y/%m/%d %H:%M:%S"*) whereas that of `{start}`, `{end}` and `{duration}` by `--time-format` (default: *%S*, absolute number of seconds). A more detailed format with `--time-format` using *%h* (hours), *%m* (minutes), *%s* (seconds) and *%i* (milliseconds) directives is possible (e.g., "%h:%m:%s.%i).

To completely disable printing detection information use `-q`.

CHAPTER 12

Save detections

You can save audio events to disk as they're detected using `-o` or `--save-detections-as`. To get a uniq file name for each event, you can use `{id}`, `{start}`, `{end}` and `{duration}` placeholders. Example:

```
auditok --save-detections-as "{id}_{start}_{end}.wav"
```

When using `{start}`, `{end}` and `{duration}` placeholders, it's recommended that the number of decimals of the corresponding values be limited to 3. You can use something like:

```
auditok -o "{id}_{start:.3f}_{end:.3f}.wav"
```

CHAPTER 13

Save whole audio stream

When reading audio data from the microphone, you most certainly want to save it to disk. For this you can use the `-O` or `--save-stream` option.

```
auditok --save-stream "stream.wav"
```

Note this will work even if you read data from another file on disk.

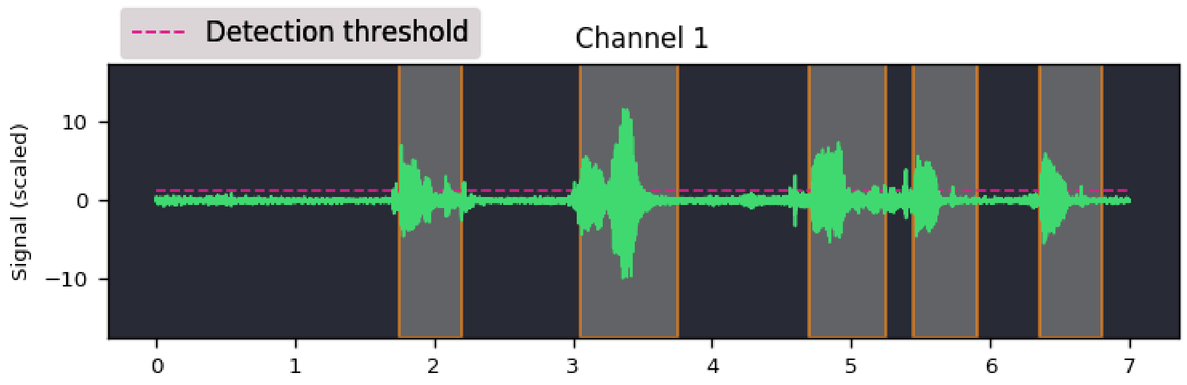
CHAPTER 14

Plot detections

Audio signal and detections can be plotted using the `-p` or `--plot` option. You can also save plot to disk using `--save-image`. The following example does both:

```
auditok -p --save-image "plot.png" # can also be 'pdf' or another image format
```

output example:



Plotting requires matplotlib.

CHAPTER 15

Core

`load`(input[, skip, max_read])	Load audio data from a source and return it as an `AudioRegion`.
`split`(input[, min_dur, max_dur, . . .])	Split audio data and return a generator of AudioRegions
`AudioRegion`(data, sampling_rate, . . . [, meta])	AudioRegion encapsulates raw audio data and provides an interface to perform simple operations on it.
`StreamTokenizer`(validator, min_length, . . .)	Class for stream tokenizers.

15.1 auditok.core.load

`auditok.core.`**`load`**(*input*, *skip=0*, *max_read=None*, ***kwargs*)

Load audio data from a source and return it as an `AudioRegion`.

Parameters

- **`input`** (`None, str, bytes, AudioSource`) – source to read audio data from. If *str*, it should be a path to a valid audio file. If *bytes*, it is used as raw audio data. If it is "-", raw data will be read from stdin. If None, read audio data from the microphone using PyAudio. If of type *bytes* or is a path to a raw audio file then *sampling_rate*, *sample_width* and *channels* parameters (or their alias) are required. If it's an `AudioSource` object it's used directly to read data.
- **`skip`** (`float, default: 0`) – amount, in seconds, of audio data to skip from source. If read from a microphone, *skip* must be 0, otherwise a *ValueError* is raised.
- **`max_read`** (`float, default: None`) – amount, in seconds, of audio data to read from source. If read from microphone, *max_read* should not be None, otherwise a *ValueError* is raised.
- **`fmt`** (`audio_format,`) – type of audio data (e.g., wav, ogg, flac, raw, etc.). This will only be used if *input* is a string path to an audio file. If not given, audio type will be guessed from file name extension or from file header.

- **sr** (*sampling_rate,*) – sampling rate of audio data. Required if *input* is a raw audio file, a *bytes* object or None (i.e., read from microphone).
- **sw** (*sample_width,*) – number of bytes used to encode one audio sample, typically 1, 2 or 4. Required for raw data, see *sampling_rate*.
- **ch** (*channels,*) – number of channels of audio data. Required for raw data, see *sampling_rate*.
- **large_file** (*bool, default: False*) – If True, AND if *input* is a path to a *wav* of a *raw* audio file (and **only** these two formats) then audio file is not fully loaded to memory in order to create the region (but the portion of data needed to create the region is of course loaded to memory). Set to True if *max_read* is significantly smaller then the size of a large audio file that shouldn't be entirely loaded to memory.

Returns region

Return type *AudioRegion*

Raises `ValueError` – raised if *input* is None (i.e., read data from microphone) and *skip* != 0 or *input* is None *max_read* is None (meaning that when reading from the microphone, no data should be skipped, and maximum amount of data to read should be explicitly provided).

15.2 auditok.core.split

`auditok.core.`**`split`**(*input*, *min_dur=0.2*, *max_dur=5*, *max_silence=0.3*, *drop_trailing_silence=False*, *strict_min_dur=False*, ***kwargs*)

Split audio data and return a generator of AudioRegions

Parameters

- **input** (*str, bytes,* AudioSource*,* AudioReader*,* AudioRegion *or None*) – input audio data. If str, it should be a path to an existing audio file. "-" is interpreted as standard input. If bytes, input is considered as raw audio data. If None, read audio from microphone. Every object that is not an *AudioReader* will be transformed into an *AudioReader* before processing. If it is an *str* that refers to a raw audio file, *bytes* or None, audio parameters should be provided using kwargs (i.e., *samplig_rate*, *sample_width* and *channels* or their alias). If *input* is str then audio format will be guessed from file extension. *audio_format* (alias *fmt*) kwarg can also be given to specify audio format explicitly. If none of these options is available, rely on backend (currently only pydub is supported) to load data.
- **min_dur** (*float, default: 0.2*) – minimun duration in seconds of a detected audio event. By using large values for *min_dur*, very short audio events (e.g., very short 1-word utterances like 'yes' or 'no') can be mis detected. Using very short values might result in a high number of short, unuseful audio events.
- **max_dur** (*float, default: 5*) – maximum duration in seconds of a detected audio event. If an audio event lasts more than *max_dur* it will be truncated. If the continuation of a truncated audio event is shorter than *min_dur* then this continuation is accepted as a valid audio event if *strict_min_dur* is False. Otherwise it is rejected.
- **max_silence** (*float, default: 0.3*) – maximum duration of continuous silence within an audio event. There might be many silent gaps of this duration within one audio event. If the continuous silence happens at the end of the event than it's kept as part of the event if *drop_trailing_silence* is False (default).

- **`drop_trailing_silence`** (`bool, default: False`) – Whether to remove trailing silence from detected events. To avoid abrupt cuts in speech, trailing silence should be kept, therefore this parameter should be False.
- **`strict_min_dur`** (`bool, default: False`) – strict minimum duration. Do not accept an audio event if it is shorter than *min_dur* even if it is contiguous to the latest valid event. This happens if the the latest detected event had reached *max_dur*.

Other Parameters

- **analysis_window, aw** (*float, default: 0.05 (50 ms)*) – duration of analysis window in seconds. A value between 0.01 (10 ms) and 0.1 (100 ms) should be good for most use-cases.
- **audio_format, fmt** (*str*) – type of audio data (e.g., wav, ogg, flac, raw, etc.). This will only be used if *input* is a string path to an audio file. If not given, audio type will be guessed from file name extension or from file header.
- **sampling_rate, sr** (*int*) – sampling rate of audio data. Required if *input* is a raw audio file, is a bytes object or None (i.e., read from microphone).
- **sample_width, sw** (*int*) – number of bytes used to encode one audio sample, typically 1, 2 or 4. Required for raw data, see *sampling_rate*.
- **channels, ch** (*int*) – number of channels of audio data. Required for raw data, see *sampling_rate*.
- **use_channel, uc** (*{None, "mix"} or int*) – which channel to use for split if *input* has multiple audio channels. Regardless of which channel is used for splitting, returned audio events contain data from *all* channels, just as *input*. The following values are accepted:
  - None (alias "any"): accept audio activity from any channel, even if other channels are silent. This is the default behavior.
  - "mix" ("avg" or "average"): mix down all channels (i.e. compute average channel) and split the resulting channel.
  - int (0 <=, > *channels*): use one channel, specified by integer id, for split.
- **large_file** (*bool, default: False*) – If True, AND if *input* is a path to a *wav* of a *raw* audio file (and only these two formats) then audio data is lazily loaded to memory (i.e., one analysis window a time). Otherwise the whole file is loaded to memory before split. Set to True if the size of the file is larger than available memory.
- **max_read, mr** (*float, default: None, read until end of stream*) – maximum data to read from source in seconds.
- **validator, val** (*callable, DataValidator*) – custom data validator. If *None* (default), an *AudioEnergyValidor* is used with the given energy threshold. Can be a callable or an instance of *DataValidator* that implements *is_valid*. In either case, it'll be called with with a window of audio data as the first parameter.
- **energy_threshold, eth** (*float, default: 50*) – energy threshold for audio activity detection. Audio regions that have enough windows of with a signal energy equal to or above this threshold are considered valid audio events. Here we are referring to this amount as the energy of the signal but to be more accurate, it is the log energy of computed as: *20 * log10(sqrt(dot(x, x) / len(x)))* (see `AudioEnergyValidator` and `calculate_energy_single_channel()`). If *validator* is given, this argument is ignored.

Yields *AudioRegion* – a generator of detected `AudioRegion` s.

15.3 auditok.core.AudioRegion

class auditok.core.**AudioRegion**(*data*, *sampling_rate*, *sample_width*, *channels*, *meta=None*)

AudioRegion encapsulates raw audio data and provides an interface to perform simple operations on it. Use *AudioRegion.load* to build an *AudioRegion* from different types of objects.

Parameters

- **data** (*bytes*) – raw audio data as a bytes object
- **sampling_rate** (*int*) – sampling rate of audio data
- **sample_width** (*int*) – number of bytes of one audio sample
- **channels** (*int*) – number of channels of audio data
- **meta** (*dict, default: None*) – any collection of <key:value> elements used to build metadata for this *AudioRegion*. Meta data can be accessed via *region.meta.key* if *key* is a valid python attribute name, or via *region.meta[key]* if not. Note that the *split()* function (or the *AudioRegion.split()* method) returns *AudioRegions* with a `start` and a `stop` meta values that indicate the location in seconds of the region in original audio data.

See also:

AudioRegion.load

__init__(*data*, *sampling_rate*, *sample_width*, *channels*, *meta=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(data, sampling_rate, sample_width, . . .)	Initialize self.
load(input[, skip, max_read])	Create an *AudioRegion* by loading data from *input*.
play([progress_bar, player])	Play audio region.
plot([scale_signal, show, figsize, save_as, . . .])	Plot audio region, one sub-plot for each channel.
save(file[, audio_format, exists_ok])	Save audio region to file.
split([min_dur, max_dur, max_silence, . . .])	Split audio region.
split_and_plot([min_dur, max_dur, . . .])	Split region and plot signal and detections.

Attributes

ch	Number of channels of audio data, alias for *channels*.
channels	Number of channels of audio data.
duration	Returns region duration in seconds.
len	Return region length in number of samples.
meta	
millis	end]``).
sample_width	Number of bytes per sample, one channel considered.
samples	Audio region as arrays of samples, one array per channel.
sampling_rate	Samling rate of audio data.
seconds	end]``).

Continued on next page

Table 3 – continued from previous page

`sr`	Samling rate of audio data, alias for *sampling_rate*.
`sw`	Number of bytes per sample, alias for *sampling_rate*.

15.4 auditok.core.StreamTokenizer

class auditok.core.**StreamTokenizer**(*validator*, *min_length*, *max_length*, *max_continuous_silence*, *init_min=0*, *init_max_silence=0*, *mode=0*)

Class for stream tokenizers. It implements a 4-state automaton scheme to extract sub-sequences of interest on the fly.

Parameters

- **validator** (callable, DataValidator (must implement *is_valid*)) – called with each data frame read from source. Should take one positional argument and return True or False for valid and invalid frames respectively.
- **min_length** (`int`) – Minimum number of frames of a valid token. This includes all tolerated non valid frames within the token.
- **max_length** (`int`) – Maximum number of frames of a valid token. This includes all tolerated non valid frames within the token.
- **max_continuous_silence** (`int`) – Maximum number of consecutive non-valid frames within a token. Note that, within a valid token, there may be many tolerated *silent* regions that contain each a number of non valid frames up to *max_continuous_silence*
- **init_min** (`int`) – Minimum number of consecutive valid frames that must be **initially** gathered before any sequence of non valid frames can be tolerated. This option is not always needed, it can be used to drop non-valid tokens as early as possible. **Default = 0** means that the option is by default ineffective.
- **init_max_silence** (`int`) – Maximum number of tolerated consecutive non-valid frames if the number already gathered valid frames has not yet reached 'init_min'.This argument is normally used if *init_min* is used. **Default = 0**, by default this argument is not taken into consideration.
- **mode** (`int`) – mode can be one of the following:

  -1 *StreamTokenizer.NORMAL* : do not drop trailing silence, and accept a token shorter than *min_length* if it is the continuation of the latest delivered token.

  -2 *StreamTokenizer.STRICT_MIN_LENGTH*: if token *i* is delivered because *max_length* is reached, and token *i+1* is immediately adjacent to token *i* (i.e. token *i* ends at frame *k* and token *i+1* starts at frame *k+1*) then accept token *i+1* only of it has a size of at least *min_length*. The default behavior is to accept token *i+1* event if it is shorter than *min_length* (provided that the above conditions are fulfilled of course).

  -3 *StreamTokenizer.DROP_TRAILING_SILENCE*: drop all tailing non-valid frames from a token to be delivered if and only if it is not **truncated**. This can be a bit tricky. A token is actually delivered if:

  – *max_continuous_silence* is reached.

  – Its length reaches *max_length*. This is referred to as a **truncated** token.

  In the current implementation, a *StreamTokenizer*'s decision is only based on already seen data and on incoming data. Thus, if a token is truncated at a non-valid but tolerated frame (*max_length* is reached but *max_continuous_silence* not yet) any tailing silence will be kept because it can potentially be part of valid token (if *max_length* was bigger). But if *max_continuous_silence* is reached before *max_length*, the delivered token will not be considered as truncated but a result of *normal* end of detection (i.e. no more valid data). In that case the trailing silence can be removed if you use the *StreamTokenizer.DROP_TRAILING_SILENCE* mode.

-4 *(StreamTokenizer.STRICT_MIN_LENGTH | StreamTokenizer.DROP_TRAILING_SILENCE)*: use both options. That means: first remove tailing silence, then check if the token still has a length of at least *min_length*.

Examples

In the following code, without *STRICT_MIN_LENGTH*, the 'BB' token is accepted although it is shorter than *min_length* (3), because it immediately follows the latest delivered token:

```
>>> from auditok.core import StreamTokenizer
>>> from StringDataSource, DataValidator
```

```
>>> class UpperCaseChecker(DataValidator):
>>>     def is_valid(self, frame):
            return frame.isupper()
>>> dsource = StringDataSource("aaaAAAABBbbb")
>>> tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
                                min_length=3,
                                max_length=4,
                                max_continuous_silence=0)
>>> tokenizer.tokenize(dsource)
[(['A', 'A', 'A', 'A'], 3, 6), (['B', 'B'], 7, 8)]
```

The following tokenizer will however reject the 'BB' token:

```
>>> dsource = StringDataSource("aaaAAAABBbbb")
>>> tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
                                min_length=3, max_length=4,
                                max_continuous_silence=0,
                                mode=StreamTokenizer.STRICT_MIN_LENGTH)
>>> tokenizer.tokenize(dsource)
[(['A', 'A', 'A', 'A'], 3, 6)]
```

```
>>> tokenizer = StreamTokenizer(
>>>                 validator=UpperCaseChecker(),
>>>                 min_length=3,
>>>                 max_length=6,
>>>                 max_continuous_silence=3,
>>>                 mode=StreamTokenizer.DROP_TRAILING_SILENCE
>>>                 )
>>> dsource = StringDataSource("aaaAAAaaaBBbbbb")
>>> tokenizer.tokenize(dsource)
[(['A', 'A', 'A', 'a', 'a', 'a'], 3, 8), (['B', 'B'], 9, 10)]
```

The first token is delivered with its tailing silence because it is truncated while the second one has its tailing frames removed.

Without *StreamTokenizer.DROP_TRAILING_SILENCE* the output would be:

```
[
    (['A', 'A', 'A', 'a', 'a', 'a'], 3, 8),
    (['B', 'B', 'b', 'b', 'b'], 9, 13)
]
```

__init__(*validator*, *min_length*, *max_length*, *max_continuous_silence*, *init_min=0*, *init_max_silence=0*, *mode=0*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(validator, min_length, max_length, . . .)	Initialize self.
`tokenize`(data_source[, callback, generator])	Read data from *data_source*, one frame a time, and process the read frames in order to detect sequences of frames that make up valid tokens.

Attributes

`DROP_TRAILING_SILENCE`
`NOISE`
`NORMAL`
`POSSIBLE_NOISE`
`POSSIBLE_SILENCE`
`SILENCE`
`STRICT_MIN_LENGTH`

`auditok.core.`**`load`**(*input*, *skip=0*, *max_read=None*, ***kwargs*)
Load audio data from a source and return it as an `AudioRegion`.

Parameters

- **input** (`None, str, bytes, AudioSource`) – source to read audio data from. If *str*, it should be a path to a valid audio file. If *bytes*, it is used as raw audio data. If it is "-", raw data will be read from stdin. If None, read audio data from the microphone using PyAudio. If of type *bytes* or is a path to a raw audio file then *sampling_rate*, *sample_width* and *channels* parameters (or their alias) are required. If it's an `AudioSource` object it's used directly to read data.
- **skip** (`float, default: 0`) – amount, in seconds, of audio data to skip from source. If read from a microphone, *skip* must be 0, otherwise a *ValueError* is raised.
- **max_read** (`float, default: None`) – amount, in seconds, of audio data to read from source. If read from microphone, *max_read* should not be None, otherwise a *ValueError* is raised.
- **fmt** (`audio_format,`) – type of audio data (e.g., wav, ogg, flac, raw, etc.). This will only be used if *input* is a string path to an audio file. If not given, audio type will be guessed from file name extension or from file header.
- **sr** (`sampling_rate,`) – sampling rate of audio data. Required if *input* is a raw audio file, a *bytes* object or None (i.e., read from microphone).
- **sw** (`sample_width,`) – number of bytes used to encode one audio sample, typically 1, 2 or 4. Required for raw data, see *sampling_rate*.

- **ch** (`channels,`) – number of channels of audio data. Required for raw data, see *sampling_rate*.
- **large_file** (`bool, default: False`) – If True, AND if *input* is a path to a *wav* of a *raw* audio file (and **only** these two formats) then audio file is not fully loaded to memory in order to create the region (but the portion of data needed to create the region is of course loaded to memory). Set to True if *max_read* is significantly smaller then the size of a large audio file that shouldn't be entirely loaded to memory.

Returns region

Return type *AudioRegion*

Raises `ValueError` – raised if *input* is None (i.e., read data from microphone) and *skip* != 0 or *input* is None *max_read* is None (meaning that when reading from the microphone, no data should be skipped, and maximum amount of data to read should be explicitly provided).

`auditok.core.`**`split`**(*input*, *min_dur=0.2*, *max_dur=5*, *max_silence=0.3*, *drop_trailing_silence=False*, *strict_min_dur=False*, ***kwargs*)

Split audio data and return a generator of AudioRegions

Parameters

- **input** (`str, bytes, AudioSource, AudioReader, AudioRegion or None`) – input audio data. If str, it should be a path to an existing audio file. "-" is interpreted as standard input. If bytes, input is considered as raw audio data. If None, read audio from microphone. Every object that is not an *AudioReader* will be transformed into an *AudioReader* before processing. If it is an *str* that refers to a raw audio file, *bytes* or None, audio parameters should be provided using kwargs (i.e., *samplig_rate*, *sample_width* and *channels* or their alias). If *input* is str then audio format will be guessed from file extension. *audio_format* (alias *fmt*) kwarg can also be given to specify audio format explicitly. If none of these options is available, rely on backend (currently only pydub is supported) to load data.
- **min_dur** (`float, default: 0.2`) – minimun duration in seconds of a detected audio event. By using large values for *min_dur*, very short audio events (e.g., very short 1-word utterances like 'yes' or 'no') can be mis detected. Using very short values might result in a high number of short, unuseful audio events.
- **max_dur** (`float, default: 5`) – maximum duration in seconds of a detected audio event. If an audio event lasts more than *max_dur* it will be truncated. If the continuation of a truncated audio event is shorter than *min_dur* then this continuation is accepted as a valid audio event if *strict_min_dur* is False. Otherwise it is rejected.
- **max_silence** (`float, default: 0.3`) – maximum duration of continuous silence within an audio event. There might be many silent gaps of this duration within one audio event. If the continuous silence happens at the end of the event than it's kept as part of the event if *drop_trailing_silence* is False (default).
- **drop_trailing_silence** (`bool, default: False`) – Whether to remove trailing silence from detected events. To avoid abrupt cuts in speech, trailing silence should be kept, therefore this parameter should be False.
- **strict_min_dur** (`bool, default: False`) – strict minimum duration. Do not accept an audio event if it is shorter than *min_dur* even if it is contiguous to the latest valid event. This happens if the the latest detected event had reached *max_dur*.

Other Parameters

- **analysis_window, aw** (*float, default: 0.05 (50 ms)*) – duration of analysis window in seconds. A value between 0.01 (10 ms) and 0.1 (100 ms) should be good for most use-cases.

- **audio_format, fmt** (*str*) – type of audio data (e.g., wav, ogg, flac, raw, etc.). This will only be used if *input* is a string path to an audio file. If not given, audio type will be guessed from file name extension or from file header.
- **sampling_rate, sr** (*int*) – sampling rate of audio data. Required if *input* is a raw audio file, is a bytes object or None (i.e., read from microphone).
- **sample_width, sw** (*int*) – number of bytes used to encode one audio sample, typically 1, 2 or 4. Required for raw data, see *sampling_rate*.
- **channels, ch** (*int*) – number of channels of audio data. Required for raw data, see *sampling_rate*.
- **use_channel, uc** (*{None, "mix"} or int*) – which channel to use for split if *input* has multiple audio channels. Regardless of which channel is used for splitting, returned audio events contain data from *all* channels, just as *input*. The following values are accepted:
  - None (alias "any"): accept audio activity from any channel, even if other channels are silent. This is the default behavior.
  - "mix" ("avg" or "average"): mix down all channels (i.e. compute average channel) and split the resulting channel.
  - int (0 <=, > *channels*): use one channel, specified by integer id, for split.
- **large_file** (*bool, default: False*) – If True, AND if *input* is a path to a *wav* of a *raw* audio file (and only these two formats) then audio data is lazily loaded to memory (i.e., one analysis window a time). Otherwise the whole file is loaded to memory before split. Set to True if the size of the file is larger than available memory.
- **max_read, mr** (*float, default: None, read until end of stream*) – maximum data to read from source in seconds.
- **validator, val** (*callable, DataValidator*) – custom data validator. If *None* (default), an *AudioEnergyValidor* is used with the given energy threshold. Can be a callable or an instance of *DataValidator* that implements *is_valid*. In either case, it'll be called with with a window of audio data as the first parameter.
- **energy_threshold, eth** (*float, default: 50*) – energy threshold for audio activity detection. Audio regions that have enough windows of with a signal energy equal to or above this threshold are considered valid audio events. Here we are referring to this amount as the energy of the signal but to be more accurate, it is the log energy of computed as: *20 * log10(sqrt(dot(x, x) / len(x)))* (see `AudioEnergyValidator` and `calculate_energy_single_channel()`). If *validator* is given, this argument is ignored.

Yields *AudioRegion* – a generator of detected `AudioRegion` s.

class `auditok.core.`**`AudioRegion`**(*data*, *sampling_rate*, *sample_width*, *channels*, *meta=None*)

AudioRegion encapsulates raw audio data and provides an interface to perform simple operations on it. Use *AudioRegion.load* to build an *AudioRegion* from different types of objects.

Parameters

- **`data`** (*`bytes`*) – raw audio data as a bytes object
- **`sampling_rate`** (*`int`*) – sampling rate of audio data
- **`sample_width`** (*`int`*) – number of bytes of one audio sample
- **`channels`** (*`int`*) – number of channels of audio data

- **meta** (*dict, default: None*) – any collection of <key:value> elements used to build metadata for this *AudioRegion*. Meta data can be accessed via *region.meta.key* if *key* is a valid python attribute name, or via *region.meta[key]* if not. Note that the `split()` function (or the `AudioRegion.split()` method) returns *AudioRegions* with a `start` and a `stop` meta values that indicate the location in seconds of the region in original audio data.

See also:

`AudioRegion.load`

ch
Number of channels of audio data, alias for *channels*.

channels
Number of channels of audio data.

duration
Returns region duration in seconds.

len
Return region length in number of samples.

classmethod load(*input*, *skip=0*, *max_read=None*, ***kwargs*)
Create an *AudioRegion* by loading data from *input*. See `load()` for parameters descripion.

Returns region

Return type *AudioRegion*

Raises `ValueError` – raised if *input* is None and *skip* != 0 or *max_read* is None.

millis
end]‘‘).

Type A view to slice audio region by milliseconds (using ‘‘region.millis[start

play(*progress_bar=False*, *player=None*, ***progress_bar_kwargs*)
Play audio region.

Parameters

- **progress_bar** (*bool, default: False*) – whether to use a progress bar while playing audio. Default: False. *progress_bar* requires *tqdm*, if not installed, no progress bar will be shown.
- **player** (*AudioPalyer, default: None*) – audio player to use. if None (default), use *player_for()* to get a new audio player.
- **progress_bar_kwargs** (*kwargs*) – keyword arguments to pass to *tqdm* progress_bar builder (e.g., use *leave=False* to clean up the screen when play finishes).

plot(*scale_signal=True*, *show=True*, *figsize=None*, *save_as=None*, *dpi=120*, *theme='auditok'*)
Plot audio region, one sub-plot for each channel.

Parameters

- **scale_signal** (*bool, default: True*) – if true, scale signal by subtracting its mean and dividing by its standard deviation before plotting.
- **show** (*bool*) – whether to show plotted signal right after the call.
- **figsize** (*tuple, default: None*) – width and height of the figure to pass to *matplotlib*.

- **save_as** (*str, default None.*) – if provided, also save plot to file.
- **dpi** (*int, default: 120*) – plot dpi to pass to *matplotlib*.
- **theme** (*str or dict, default: "auditok"*) – plot theme to use. Currently only "auditok" theme is implemented. To provide you own them see `auditok.plotting.AUDITOK_PLOT_THEME`.

sample_width
Number of bytes per sample, one channel considered.

samples
Audio region as arrays of samples, one array per channel.

sampling_rate
Samling rate of audio data.

save (*file*, *audio_format=None*, *exists_ok=True*, ***audio_parameters*)
Save audio region to file.

Parameters

- **file** (*str*) – path to output audio file. May contain *{duration}* placeholder as well as any place holder that this region's metadata might contain (e.g., regions returned by *split* contain metadata with *start* and *end* attributes that can be used to build output file name as *{meta.start}* and *{meta.end}*. See examples using placeholders with formatting.
- **audio_format** (*str, default: None*) – format used to save audio data. If None (default), format is guessed from file name's extension. If file name has no extension, audio data is saved as a raw (headerless) audio file.
- **exists_ok** (*bool, default: True*) – If True, overwrite *file* if a file with the same name exists. If False, raise an *IOError* if *file* exists.
- **audio_parameters** (*dict*) – any keyword arguments to be passed to audio saving backend.

Returns

- **file** (*str*) – name of output file with replaced placehoders.
- *Raises* – IOError if *file* exists and *exists_ok* is False.

Examples

```
>>> region = AudioRegion(b'\0' * 2 * 24000,
>>>                      sampling_rate=16000,
>>>                      sample_width=2,
>>>                      channels=1)
>>> region.meta.start = 2.25
>>> region.meta.end = 2.25 + region.duration
>>> region.save('audio_{meta.start}-{meta.end}.wav')
>>> audio_2.25-3.75.wav
>>> region.save('region_{meta.start:.3f}_{duration:.3f}.wav')
audio_2.250_1.500.wav
```

seconds
end]‘‘).

Type A view to slice audio region by seconds (using ‘‘region.seconds[start

split(*min_dur=0.2*, *max_dur=5*, *max_silence=0.3*, *drop_trailing_silence=False*, *strict_min_dur=False*, ***kwargs*)
Split audio region. See `auditok.split()` for a comprehensive description of split parameters. See Also `AudioRegio.split_and_plot()`.

split_and_plot(*min_dur=0.2*, *max_dur=5*, *max_silence=0.3*, *drop_trailing_silence=False*, *strict_min_dur=False*, *scale_signal=True*, *show=True*, *figsize=None*, *save_as=None*, *dpi=120*, *theme='auditok'*, ***kwargs*)
Split region and plot signal and detections. Alias: `splitp()`. See `auditok.split()` for a comprehensive description of split parameters. Also see `plot()` for plot parameters.

sr
Samling rate of audio data, alias for *sampling_rate*.

sw
Number of bytes per sample, alias for *sampling_rate*.

class auditok.core.**StreamTokenizer**(*validator*, *min_length*, *max_length*, *max_continuous_silence*, *init_min=0*, *init_max_silence=0*, *mode=0*)
Class for stream tokenizers. It implements a 4-state automaton scheme to extract sub-sequences of interest on the fly.

Parameters

- **validator** (callable, DataValidator (must implement *is_valid*)) – called with each data frame read from source. Should take one positional argument and return True or False for valid and invalid frames respectively.
- **min_length** (*int*) – Minimum number of frames of a valid token. This includes all tolerated non valid frames within the token.
- **max_length** (*int*) – Maximum number of frames of a valid token. This includes all tolerated non valid frames within the token.
- **max_continuous_silence** (*int*) – Maximum number of consecutive non-valid frames within a token. Note that, within a valid token, there may be many tolerated *silent* regions that contain each a number of non valid frames up to *max_continuous_silence*
- **init_min** (*int*) – Minimum number of consecutive valid frames that must be **initially** gathered before any sequence of non valid frames can be tolerated. This option is not always needed, it can be used to drop non-valid tokens as early as possible. **Default = 0** means that the option is by default ineffective.
- **init_max_silence** (*int*) – Maximum number of tolerated consecutive non-valid frames if the number already gathered valid frames has not yet reached 'init_min'.This argument is normally used if *init_min* is used. **Default = 0**, by default this argument is not taken into consideration.
- **mode** (*int*) – mode can be one of the following:

  -1 *StreamTokenizer.NORMAL* : do not drop trailing silence, and accept a token shorter than *min_length* if it is the continuation of the latest delivered token.

  -2 *StreamTokenizer.STRICT_MIN_LENGTH*: if token *i* is delivered because *max_length* is reached, and token *i+1* is immediately adjacent to token *i* (i.e. token *i* ends at frame *k* and token *i+1* starts at frame *k+1*) then accept token *i+1* only of it has a size of at least *min_length*. The default behavior is to accept token *i+1* event if it is shorter than *min_length* (provided that the above conditions are fulfilled of course).

  -3 *StreamTokenizer.DROP_TRAILING_SILENCE*: drop all tailing non-valid frames from a token to be delivered if and only if it is not **truncated**. This can be a bit tricky.

A token is actually delivered if:

- *max_continuous_silence* is reached.
- Its length reaches *max_length*. This is referred to as a **truncated** token.

In the current implementation, a *StreamTokenizer*'s decision is only based on already seen data and on incoming data. Thus, if a token is truncated at a non-valid but tolerated frame (*max_length* is reached but *max_continuous_silence* not yet) any tailing silence will be kept because it can potentially be part of valid token (if *max_length* was bigger). But if *max_continuous_silence* is reached before *max_length*, the delivered token will not be considered as truncated but a result of *normal* end of detection (i.e. no more valid data). In that case the trailing silence can be removed if you use the *StreamTokenizer.DROP_TRAILING_SILENCE* mode.

-4 *(StreamTokenizer.STRICT_MIN_LENGTH | StreamTokenizer.DROP_TRAILING_SILENCE)*: use both options. That means: first remove tailing silence, then check if the token still has a length of at least *min_length*.

Examples

In the following code, without *STRICT_MIN_LENGTH*, the 'BB' token is accepted although it is shorter than *min_length* (3), because it immediately follows the latest delivered token:

```
>>> from auditok.core import StreamTokenizer
>>> from StringDataSource, DataValidator
```

```
>>> class UpperCaseChecker(DataValidator):
>>>     def is_valid(self, frame):
            return frame.isupper()
>>> dsource = StringDataSource("aaaAAAABBbbb")
>>> tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
                                min_length=3,
                                max_length=4,
                                max_continuous_silence=0)
>>> tokenizer.tokenize(dsource)
[(['A', 'A', 'A', 'A'], 3, 6), (['B', 'B'], 7, 8)]
```

The following tokenizer will however reject the 'BB' token:

```
>>> dsource = StringDataSource("aaaAAAABBbbb")
>>> tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
                                min_length=3, max_length=4,
                                max_continuous_silence=0,
                                mode=StreamTokenizer.STRICT_MIN_LENGTH)
>>> tokenizer.tokenize(dsource)
[(['A', 'A', 'A', 'A'], 3, 6)]
```

```
>>> tokenizer = StreamTokenizer(
>>>                 validator=UpperCaseChecker(),
>>>                 min_length=3,
>>>                 max_length=6,
>>>                 max_continuous_silence=3,
>>>                 mode=StreamTokenizer.DROP_TRAILING_SILENCE
>>>                 )
>>> dsource = StringDataSource("aaaAAAaaaBBbbbb")
```

(continues on next page)

(continued from previous page)

```
>>> tokenizer.tokenize(dsource)
[(['A', 'A', 'A', 'a', 'a', 'a'], 3, 8), (['B', 'B'], 9, 10)]
```

The first token is delivered with its tailing silence because it is truncated while the second one has its tailing frames removed.

Without *StreamTokenizer.DROP_TRAILING_SILENCE* the output would be:

```
[
    (['A', 'A', 'A', 'a', 'a', 'a'], 3, 8),
    (['B', 'B', 'b', 'b', 'b'], 9, 13)
]
```

tokenize(*data_source*, *callback=None*, *generator=False*)

Read data from *data_source*, one frame a time, and process the read frames in order to detect sequences of frames that make up valid tokens.

Parameters

data_source [instance of the `DataSource` class that] implements a *read* method. 'read' should return a slice of signal, i.e. frame (of whatever type as long as it can be processed by validator) and None if there is no more signal.

callback [an optional 3-argument function.] If a *callback* function is given, it will be called each time a valid token is found.

Returns A list of tokens if *callback* is None. Each token is tuple with the following elements:

where *data* is a list of read frames, *start*: index of the first frame in the original data and *end* : index of the last frame.

CHAPTER 16

Util

AudioEnergyValidator(energy_threshold, . . .)	A validator based on audio signal energy.
AudioReader(input[, block_dur, hop_dur, . . .])	Class to read fixed-size chunks of audio data from a source.
Recorder(input[, block_dur, hop_dur, max_read])	Class to read fixed-size chunks of audio data from a source and keeps data in a cache.
make_duration_formatter(fmt)	Make and return a function used to format durations in seconds.
make_channel_selector(sample_width, channels)	Create and return a callable used for audio channel selection.

16.1 auditok.util.AudioEnergyValidator

class auditok.util.**AudioEnergyValidator**(*energy_threshold*, *sample_width*, *channels*, *use_channel=None*)

A validator based on audio signal energy. For an input window of N audio samples (see *AudioEnergyValidator.is_valid()*), the energy is computed as:

$$energy = 20\log(\sqrt(1/N\sum_i^N {a_i}^2))wherea\_iisthei-thaudiosample.$$

Parameters

- **energy_threshold** (*float*) – minimum energy that audio window should have to be valid.
- **sample_width** (*int*) – size in bytes of one audio sample.
- **channels** (*int*) – number of channels of audio data.
- **use_channel** (*{None, "any", "mix", "avg", "average"} or int*) – channel to use for energy computation. The following values are accepted:
  - None (alias “any”) : compute energy for each of the channels and return the maximum value.
  - ”mix” (alias “avg” or “average”) : compute the average channel then compute its energy.
  - int (>= 0 , < *channels*) : compute the energy of the specified channel and ignore the other ones.

Returns
: **energy** – energy of the audio window.

Return type
: float

`__init__`(*energy_threshold*, *sample_width*, *channels*, *use_channel=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(energy_threshold, sample_width, . . .)	Initialize self.
`is_valid`(data)	**param data** array of raw audio data

16.2 auditok.util.AudioReader

class `auditok.util.`**`AudioReader`**(*input*, *block_dur=0.01*, *hop_dur=None*, *record=False*, *max_read=None*, ***kwargs*)

Class to read fixed-size chunks of audio data from a source. A source can be a file on disk, standard input (with *input* = "-") or microphone. This is normally used by tokenization algorithms that expect source objects with a *read* function that returns a windows of data of the same size at each call expect when remaining data does not make up a full window.

Objects of this class can be set up to return audio windows with a given overlap and to record the whole stream for later access (useful when reading data from the microphone). They can also have a limit for the maximum amount of data to read.

Parameters

- **input** (*str, bytes,* `AudioSource`, `AudioReader`, `AudioRegion` *or None*) – input audio data. If the type of the passed argument is *str*, it should be a path to an existing audio file. "-" is interpreted as standardinput. If the type is *bytes*, input is considered as a buffer of raw audio data. If None, read audio from microphone. Every object that is not an `AudioReader` will be transformed, when possible, into an `AudioSource` before processing. If it is an *str* that refers to a raw audio file, *bytes* or None, audio parameters should be provided using kwargs (i.e., *samplig_rate*, *sample_width* and *channels* or their alias).
- **block_dur** (*float, default: 0.01*) – length in seconds of audio windows to return at each *read* call.
- **hop_dur** (*float, default: None*) – length in seconds of data amount to skip from previous window. If defined, it is used to compute the temporal overlap between previous and current window (nameply *overlap = block_dur - hop_dur*). Default, None, means that consecutive windows do not overlap.
- **record** (*bool, default: False*) – whether to record read audio data for later access. If True, audio data can be retrieved by first calling *rewind()*, then using the *data* property. Note that once *rewind()* is called, no new data will be read from source (subsequent *read()* call will read data from cache) and that there's no need to call *rewind()* again to access *data* property.
- **max_read** (*float, default: None*) – maximum amount of audio data to read in seconds. Default is None meaning that data will be read until end of stream is reached or,

when reading from microphone a Ctrl-C is sent.

- **input is None, of type bytes or a raw audio files some of the** (*When*) –
- **kwargs are mandatory.** (*follwing*) –

Other Parameters

- **audio_format, fmt** (*str*) – type of audio data (e.g., wav, ogg, flac, raw, etc.). This will only be used if *input* is a string path to an audio file. If not given, audio type will be guessed from file name extension or from file header.
- **sampling_rate, sr** (*int*) – sampling rate of audio data. Required if *input* is a raw audio file, is a bytes object or None (i.e., read from microphone).
- **sample_width, sw** (*int*) – number of bytes used to encode one audio sample, typically 1, 2 or 4. Required for raw data, see *sampling_rate*.
- **channels, ch** (*int*) – number of channels of audio data. Required for raw data, see *sampling_rate*.
- **use_channel, uc** (*{None, "any", "mix", "avg", "average"} or int*) – which channel to use for split if *input* has multiple audio channels. Regardless of which channel is used for splitting, returned audio events contain data from *all* the channels of *input*. The following values are accepted:
  - None (alias "any"): accept audio activity from any channel, even if other channels are silent. This is the default behavior.
  - "mix" (alias "avg" or "average"): mix down all channels (i.e., compute average channel) and split the resulting channel.
  - int (>= 0 , < *channels*): use one channel, specified by its integer id, for split.
- **large_file** (*bool, default: False*) – If True, AND if *input* is a path to a *wav* of a *raw* audio file (and only these two formats) then audio data is lazily loaded to memory (i.e., one analysis window a time). Otherwise the whole file is loaded to memory before split. Set to True if the size of the file is larger than available memory.

__init__(*input*, *block_dur=0.01*, *hop_dur=None*, *record=False*, *max_read=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(input[, block_dur, hop_dur, ...])	Initialize self.
`read`()	Read a block (i.e., window) of data read from this source.

Attributes

`block_dur`
`hop_dur`
`hop_size`
`max_read`
`rewindable`

16.3 auditok.util.Recorder

class auditok.util.**Recorder**(*input*, *block_dur=0.01*, *hop_dur=None*, *max_read=None*, ***kwargs*)

Class to read fixed-size chunks of audio data from a source and keeps data in a cache. Using this class is equivalent to initializing `AudioReader` with *record=True*. For more information about the other parameters see `AudioReader`.

Once the desired amount of data is read, you can call the `rewind()` method then get the recorded data via the `data` attribute. You can also re-read cached data one window a time by calling `read()`.

__init__(*input*, *block_dur=0.01*, *hop_dur=None*, *max_read=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(input[, block_dur, hop_dur, max_read])	Initialize self.
read()	Read a block (i.e., window) of data read from this source.

Attributes

`block_dur`
`hop_dur`
`hop_size`
`max_read`
`rewindable`

16.4 auditok.util.make_duration_formatter

auditok.util.**make_duration_formatter**(*fmt*)

Make and return a function used to format durations in seconds. Accepted format directives are:

- `%S` : absolute number of seconds with 3 decimals. This direction should be used alone.
- `%i` : milliseconds
- `%s` : seconds
- `%m` : minutes
- `%h` : hours

These last 4 directives should all be specified. They can be placed anywhere in the input string.

Parameters **fmt** (`str`) – duration format.

Returns **formatter** – a function that takes a duration in seconds (float) and returns a string that corresponds to that duration.

Return type callable

Raises `TimeFormatError` – if the format contains an unknown directive.

Examples

Using `%S`:

```
formatter = make_duration_formatter("%S")
formatter(123.589)
'123.589'
formatter(123)
'123.000'
```

Using the other directives:

```
formatter = make_duration_formatter("%h:%m:%s.%i")
formatter(3600+120+3.25)
'01:02:03.250'

formatter = make_duration_formatter("%h hrs, %m min, %s sec and %i ms")
formatter(3600+120+3.25)
'01 hrs, 02 min, 03 sec and 250 ms'

# omitting one of the 4 directives might result in a wrong duration
formatter = make_duration_formatter("%m min, %s sec and %i ms")
formatter(3600+120+3.25)
'02 min, 03 sec and 250 ms'
```

16.5 auditok.util.make_channel_selector

`auditok.util.`**`make_channel_selector`**(*sample_width*, *channels*, *selected=None*)

Create and return a callable used for audio channel selection. The returned selector can be used as *selector(audio_data)* and returns data that contains selected channel only.

Importantly, if *selected* is None or equals "any", *selector(audio_data)* will separate and return a list of available channels: *[data_channe_1, data_channe_2, . . .].*

Note also that returned *selector* expects *bytes* format for input data but does notnecessarily return a *bytes* object. In fact, in order to extract the desired channel (or compute the average channel if *selected* = "avg"), it first converts input data into a *array.array* (or *numpy.ndarray*) object. After channel of interst is selected/computed, it is returned as such, without any reconversion to *bytes*. This behavior is wanted for efficiency purposes because returned objects can be directly used as buffers of bytes. In any case, returned objects can be converted back to *bytes* using *bytes(obj)*.

Exception to this is the special case where *channels* = 1 in which input data is returned without any processing.

Parameters

- **`sample_width`** (*`int`*) – number of bytes used to encode one audio sample, should be 1, 2 or 4.
- **`channels`** (*`int`*) – number of channels of raw audio data that the returned selector should expect.
- **`selected`** (*`int or str, default: None`*) – audio channel to select and return when calling *selector(raw_data)*. It should be an int >= *-channels* and < *channels*. If one of "mix", "avg" or "average" is passed then *selector* will return the average channel of audio data. If None or "any", return a list of all available channels at each call.

Returns **selector** – a callable that can be used as *selector(audio_data)* and returns data that contains channel of interst.

Return type callable

Raises `ValueError` – if *sample_width* is not one of 1, 2 or 4, or if *selected* has an unexpected value.

`auditok.util.`**`make_duration_formatter`**(*fmt*)

Make and return a function used to format durations in seconds. Accepted format directives are:

- `%S` : absolute number of seconds with 3 decimals. This direction should be used alone.
- `%i` : milliseconds
- `%s` : seconds
- `%m` : minutes
- `%h` : hours

These last 4 directives should all be specified. They can be placed anywhere in the input string.

Parameters **`fmt`** (*`str`*) – duration format.

Returns **formatter** – a function that takes a duration in seconds (float) and returns a string that corresponds to that duration.

Return type callable

Raises `TimeFormatError` – if the format contains an unknown directive.

Examples

Using `%S`:

```
formatter = make_duration_formatter("%S")
formatter(123.589)
'123.589'
formatter(123)
'123.000'
```

Using the other directives:

```
formatter = make_duration_formatter("%h:%m:%s.%i")
formatter(3600+120+3.25)
'01:02:03.250'

formatter = make_duration_formatter("%h hrs, %m min, %s sec and %i ms")
formatter(3600+120+3.25)
'01 hrs, 02 min, 03 sec and 250 ms'

# omitting one of the 4 directives might result in a wrong duration
formatter = make_duration_formatter("%m min, %s sec and %i ms")
formatter(3600+120+3.25)
'02 min, 03 sec and 250 ms'
```

`auditok.util.`**`make_channel_selector`**(*sample_width*, *channels*, *selected=None*)

Create and return a callable used for audio channel selection. The returned selector can be used as *selector(audio_data)* and returns data that contains selected channel only.

Importantly, if *selected* is None or equals “any”, *selector(audio_data)* will separate and return a list of available channels: *[data_channe_1, data_channe_2, …]*.

Note also that returned *selector* expects *bytes* format for input data but does notnecessarily return a *bytes* object. In fact, in order to extract the desired channel (or compute the average channel if *selected* = "avg"), it first converts input data into a *array.array* (or *numpy.ndarray*) object. After channel of interst is selected/computed, it is returned as such, without any reconversion to *bytes*. This behavior is wanted for efficiency purposes because returned objects can be directly used as buffers of bytes. In any case, returned objects can be converted back to *bytes* using *bytes(obj)*.

Exception to this is the special case where *channels* = 1 in which input data is returned without any processing.

Parameters

- **`sample_width`** (`int`) – number of bytes used to encode one audio sample, should be 1, 2 or 4.
- **`channels`** (`int`) – number of channels of raw audio data that the returned selector should expect.
- **`selected`** (`int or str, default: None`) – audio channel to select and return when calling *selector(raw_data)*. It should be an int >= -*channels* and < *channels*. If one of "mix", "avg" or "average" is passed then *selector* will return the average channel of audio data. If None or "any", return a list of all available channels at each call.

Returns **selector** – a callable that can be used as *selector(audio_data)* and returns data that contains channel of interst.

Return type callable

Raises `ValueError` – if *sample_width* is not one of 1, 2 or 4, or if *selected* has an unexpected value.

class `auditok.util.`**`DataSource`**

Base class for objects passed to `StreamTokenizer.tokenize()`. Subclasses should implement a *`DataSource.read()`* method.

`read`()

Read a block (i.e., window) of data read from this source. If no more data is available, return None.

class `auditok.util.`**`DataValidator`**

Base class for a validator object used by *`core.StreamTokenizer`* to check if read data is valid. Subclasses should implement *`is_valid()`* method.

`is_valid`(*data*)

Check whether *data* is valid

class `auditok.util.`**`StringDataSource`**(*data*)

Class that represent a *`DataSource`* as a string buffer. Each call to *`DataSource.read()`* returns on character and moves one step forward. If the end of the buffer is reached, *`read()`* returns None.

Parameters **`data`** (`str`) – a string object used as data.

`read`()

Read one character from buffer.

Returns **char** – current character or None if end of buffer is reached.

Return type str

`set_data`(*data*)

Set a new data buffer.

Parameters **`data`** (`str`) – new data buffer.

class auditok.util.**ADSFactory**

Deprecated since version 2.0.0: *ADSFactory* will be removed in auditok 2.0.1, use instances of `AudioReader` instead.

Factory class that makes it easy to create an `AudioDataSource` object that implements `DataSource` and can therefore be passed to `auditok.core.StreamTokenizer.tokenize()`.

Whether you read audio data from a file, the microphone or a memory buffer, this factory instantiates and returns the right `AudioDataSource` object.

There are many other features you want a `AudioDataSource` object to have, such as: memorize all read audio data so that you can rewind and reuse it (especially useful when reading data from the microphone), read a fixed amount of data (also useful when reading from the microphone), read overlapping audio frames (often needed when dosing a spectral analysis of data).

`ADSFactory.ads()` automatically creates and return object with the desired behavior according to the supplied keyword arguments.

static ads(***kwargs*)

Create an return an `AudioDataSource`. The type and behavior of the object is the result of the supplied parameters. Called without any parameters, the class will read audio data from the available built-in microphone with the default parameters.

Parameters

- **sr** (*sampling_rate,*) – number of audio samples per second of input audio stream.
- **sw** (*sample_width,*) – number of bytes per sample, must be one of 1, 2 or 4
- **ch** (*channels,*) – number of audio channels, only a value of 1 is currently accepted.
- **fpb** (*frames_per_buffer,*) – number of samples of PyAudio buffer.
- **asrc** (*audio_source,*) – *AudioSource* to read data from
- **fn** (*filename,*) – create an *AudioSource* object using this file
- **db** (*data_buffer,*) – build an *io.BufferAudioSource* using data in *data_buffer*. If this keyword is used, *sampling_rate*, *sample_width* and *channels* are passed to *io.BufferAudioSource* constructor and used instead of default values.
- **mt** (*max_time,*) – maximum time (in seconds) to read. Default behavior: read until there is no more data available.
- **rec** (*record,*) – save all read data in cache. Provide a navigable object which has a *rewind* method.
- **bd** (*block_dur,*) – processing block duration in seconds. This represents the quantity of audio data to return each time the `read()` method is invoked. If *block_dur* is 0.025 (i.e. 25 ms) and the sampling rate is 8000 and the sample width is 2 bytes, `read()` returns a buffer of 0.025 * 8000 * 2 = 400 bytes at most. This parameter will be looked for (and used if available) before *block_size*. If neither parameter is given, *block_dur* will be set to 0.01 second (i.e. 10 ms)
- **hd** (*hop_dur,*) – quantity of data to skip from current processing window. if *hop_dur* is supplied then there will be an overlap of *block_dur* - *hop_dur* between two adjacent blocks. This parameter will be looked for (and used if available) before *hop_size*. If neither parameter is given, *hop_dur* will be set to *block_dur* which means that there will be no overlap between two consecutively read blocks.
- **bs** (*block_size,*) – number of samples to read each time the *read* method is called. Default: a block size that represents a window of 10ms, so for a sampling rate of 16000, the default *block_size* is 160 samples, for a rate of 44100, *block_size* = 441 samples, etc.

- **hs** (*hop_size,*) – determines the number of overlapping samples between two adjacent read windows. For a *hop_size* of value *N*, the overlap is *block_size* - *N*. Default : *hop_size* = *block_size*, means that there is no overlap.

Returns **audio_data_source** – an *AudioDataSource* object build with input parameters.

Return type AudioDataSource

`auditok.util.`**`AudioDataSource`**

alias of *auditok.util.AudioReader*

class `auditok.util.`**`AudioReader`**(*input*, *block_dur=0.01*, *hop_dur=None*, *record=False*, *max_read=None*, ***kwargs*)

Class to read fixed-size chunks of audio data from a source. A source can be a file on disk, standard input (with *input* = "-") or microphone. This is normally used by tokenization algorithms that expect source objects with a *read* function that returns a windows of data of the same size at each call expect when remaining data does not make up a full window.

Objects of this class can be set up to return audio windows with a given overlap and to record the whole stream for later access (useful when reading data from the microphone). They can also have a limit for the maximum amount of data to read.

Parameters

- **input** (*str, bytes, AudioSource, AudioReader, AudioRegion or None*) – input audio data. If the type of the passed argument is *str*, it should be a path to an existing audio file. "-" is interpreted as standardinput. If the type is *bytes*, input is considered as a buffer of raw audio data. If None, read audio from microphone. Every object that is not an *AudioReader* will be transformed, when possible, into an `AudioSource` before processing. If it is an *str* that refers to a raw audio file, *bytes* or None, audio parameters should be provided using kwargs (i.e., *samplig_rate*, *sample_width* and *channels* or their alias).
- **block_dur** (*float, default: 0.01*) – length in seconds of audio windows to return at each *read* call.
- **hop_dur** (*float, default: None*) – length in seconds of data amount to skip from previous window. If defined, it is used to compute the temporal overlap between previous and current window (nameply *overlap* = *block_dur* - *hop_dur*). Default, None, means that consecutive windows do not overlap.
- **record** (*bool, default: False*) – whether to record read audio data for later access. If True, audio data can be retrieved by first calling *rewind()*, then using the *data* property. Note that once *rewind()* is called, no new data will be read from source (subsequent *read()* call will read data from cache) and that there's no need to call *rewind()* again to access *data* property.
- **max_read** (*float, default: None*) – maximum amount of audio data to read in seconds. Default is None meaning that data will be read until end of stream is reached or, when reading from microphone a Ctrl-C is sent.
- **input is None, of type bytes or a raw audio files some of the** (*When*) –
- **kwargs are mandatory.** (*follwing*) –

Other Parameters

- **audio_format, fmt** (*str*) – type of audio data (e.g., wav, ogg, flac, raw, etc.). This will only be used if *input* is a string path to an audio file. If not given, audio type will be guessed from file name extension or from file header.

- **sampling_rate, sr** (*int*) – sampling rate of audio data. Required if *input* is a raw audio file, is a bytes object or None (i.e., read from microphone).
- **sample_width, sw** (*int*) – number of bytes used to encode one audio sample, typically 1, 2 or 4. Required for raw data, see *sampling_rate*.
- **channels, ch** (*int*) – number of channels of audio data. Required for raw data, see *sampling_rate*.
- **use_channel, uc** (*{None, "any", "mix", "avg", "average"} or int*) – which channel to use for split if *input* has multiple audio channels. Regardless of which channel is used for splitting, returned audio events contain data from *all* the channels of *input*. The following values are accepted:
  - None (alias "any"): accept audio activity from any channel, even if other channels are silent. This is the default behavior.
  - "mix" (alias "avg" or "average"): mix down all channels (i.e., compute average channel) and split the resulting channel.
  - int (>= 0 , < *channels*): use one channel, specified by its integer id, for split.
- **large_file** (*bool, default: False*) – If True, AND if *input* is a path to a *wav* of a *raw* audio file (and only these two formats) then audio data is lazily loaded to memory (i.e., one analysis window a time). Otherwise the whole file is loaded to memory before split. Set to True if the size of the file is larger than available memory.

read()

Read a block (i.e., window) of data read from this source. If no more data is available, return None.

class auditok.util.**Recorder**(*input*, *block_dur=0.01*, *hop_dur=None*, *max_read=None*, ***kwargs*)

Class to read fixed-size chunks of audio data from a source and keeps data in a cache. Using this class is equivalent to initializing `AudioReader` with *record=True*. For more information about the other parameters see `AudioReader`.

Once the desired amount of data is read, you can call the `rewind()` method then get the recorded data via the `data` attribute. You can also re-read cached data one window a time by calling `read()`.

class auditok.util.**AudioEnergyValidator**(*energy_threshold*, *sample_width*, *channels*, *use_channel=None*)

A validator based on audio signal energy. For an input window of *N* audio samples (see `AudioEnergyValidator.is_valid()`), the energy is computed as:

$$energy = 20\log(\sqrt(1/N\sum_{i}^{N} {a_i}^2))wherea\_iisthei-thaudiosample.$$

Parameters

- **`energy_threshold`** (*`float`*) – minimum energy that audio window should have to be valid.
- **`sample_width`** (*`int`*) – size in bytes of one audio sample.
- **`channels`** (*`int`*) – number of channels of audio data.
- **`use_channel`** (*`{None, "any", "mix", "avg", "average"} or int`*) – channel to use for energy computation. The following values are accepted:
  - None (alias "any") : compute energy for each of the channels and return the maximum value.
  - "mix" (alias "avg" or "average") : compute the average channel then compute its energy.
  - int (>= 0 , < *channels*) : compute the energy of the specified channel and ignore the other ones.

Returns

energy – energy of the audio window.

Return type
float

`is_valid`(*data*)

Parameters **`data`** (*`bytes-like`*) – array of raw audio data

Returns True if the energy of audio data is >= threshold, False otherwise.

Return type bool

CHAPTER 17

Low-level IO

Module for low-level audio input-output operations.

AudioSource(sampling_rate, sample_width, . . .)	Base class for audio source objects.
Rewindable(sampling_rate, sample_width, channels)	Base class for rewindable audio streams.
BufferAudioSource(data[, sampling_rate, . . .])	An *AudioSource* that encapsulates and reads data from a memory buffer.
WaveAudioSource(filename)	A class for an *AudioSource* that reads data from a wave file.
PyAudioSource([sampling_rate, sample_width, . . .])	A class for an *AudioSource* that reads data from built-in microphone using PyAudio (https://people.csail.mit.edu/hubert/pyaudio/).
StdinAudioSource([sampling_rate, . . .])	A class for an *AudioSource* that reads data from standard input.
PyAudioPlayer([sampling_rate, sample_width, . . .])	A class for audio playback using Pyaudio (https://people.csail.mit.edu/hubert/pyaudio/).
from_file(filename[, audio_format, large_file])	Read audio data from *filename* and return an *AudioSource* object.
to_file(data, file[, audio_format])	Writes audio data to file.
player_for(source)	Return an *AudioPlayer* compatible with *source* (i.e., has the same sampling rate, sample width and number of channels).

17.1 auditok.io.AudioSource

class auditok.io.**AudioSource**(*sampling_rate*, *sample_width*, *channels*)

Base class for audio source objects.

Subclasses should implement methods to open/close and audio stream and read the desired amount of audio samples.

Parameters

- **sampling_rate** (*int*) – number of samples per second of audio data.
- **sample_width** (*int*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **channels** (*int*) – number of channels of audio data.

__init__(*sampling_rate*, *sample_width*, *channels*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(sampling_rate, sample_width, channels)	Initialize self.
`close`()	Close audio source.
`is_open`()	Return True if audio source is open, False otherwise.
`open`()	Open audio source.
`read`(size)	Read and return *size* audio samples at most.

Attributes

`ch`	Number of channels in audio stream (alias for *channels*).
`channels`	Number of channels in audio stream.
`sample_width`	Number of bytes used to represent one audio sample.
`sampling_rate`	Number of samples per second of audio stream.
`sr`	Number of samples per second of audio stream (alias for *sampling_rate*).
`sw`	Number of bytes used to represent one audio sample (alias for *sample_width*).

17.2 auditok.io.Rewindable

class auditok.io.**Rewindable**(*sampling_rate*, *sample_width*, *channels*)
Base class for rewindable audio streams.

Subclasses should implement a method to return back to the start of an the stream (*rewind*), as well as a property getter/setter named *position* that reads/sets stream position expressed in number of samples.

__init__(*sampling_rate*, *sample_width*, *channels*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(sampling_rate, sample_width, channels)	Initialize self.
`close`()	Close audio source.
`is_open`()	Return True if audio source is open, False otherwise.
`open`()	Open audio source.

Continued on next page

Table 4 – continued from previous page

`read(size)`	Read and return *size* audio samples at most.
`rewind()`	Go back to the beginning of audio stream.

Attributes

`ch`	Number of channels in audio stream (alias for *channels*).
`channels`	Number of channels in audio stream.
`position`	Return stream position in number of samples.
`position_ms`	Return stream position in milliseconds.
`position_s`	Return stream position in seconds.
`sample_width`	Number of bytes used to represent one audio sample.
`sampling_rate`	Number of samples per second of audio stream.
`sr`	Number of samples per second of audio stream (alias for *sampling_rate)*.
`sw`	Number of bytes used to represent one audio sample (alias for *sample_width*).

17.3 auditok.io.BufferAudioSource

class auditok.io.**BufferAudioSource**(*data*, *sampling_rate=16000*, *sample_width=2*, *channels=1*)

An *AudioSource* that encapsulates and reads data from a memory buffer.

This class implements the *Rewindable* interface. :param data: audio data :type data: bytes :param sampling_rate: number of samples per second of audio data. :type sampling_rate: int, default: 16000 :param sample_width: size in bytes of one audio sample. Possible values: 1, 2 or 4. :type sample_width: int, default: 2 :param channels: number of channels of audio data. :type channels: int, default: 1

__init__(*data*, *sampling_rate=16000*, *sample_width=2*, *channels=1*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__(data[, sampling_rate, …])`	Initialize self.
`close()`	Close audio source.
`is_open()`	Return True if audio source is open, False otherwise.
`open()`	Open audio source.
`read(size)`	Read and return *size* audio samples at most.
`rewind()`	Go back to the beginning of audio stream.

Attributes

`ch`	Number of channels in audio stream (alias for *channels*).
`channels`	Number of channels in audio stream.
`data`	Get raw audio data as a *bytes* object.
`position`	Get stream position in number of samples

Continued on next page

Table 7 – continued from previous page

`position_ms`	Get stream position in milliseconds.
`position_s`	Return stream position in seconds.
`sample_width`	Number of bytes used to represent one audio sample.
`sampling_rate`	Number of samples per second of audio stream.
`sr`	Number of samples per second of audio stream (alias for *sampling_rate)*.
`sw`	Number of bytes used to represent one audio sample (alias for *sample_width*).

17.4 auditok.io.WaveAudioSource

class `auditok.io.`**`WaveAudioSource`**(*filename*)

A class for an *AudioSource* that reads data from a wave file.

This class should be used for large wave files to avoid loading the whole data to memory.

Parameters **`filename`** (`str`) – path to a valid wave file.

`__init__`(*filename*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename)	Initialize self.
`close`()	Close audio source.
`is_open`()	Return True if audio source is open, False otherwise.
`open`()	Open audio source.
`read`(size)	Read and return *size* audio samples at most.

Attributes

`ch`	Number of channels in audio stream (alias for *channels*).
`channels`	Number of channels in audio stream.
`sample_width`	Number of bytes used to represent one audio sample.
`sampling_rate`	Number of samples per second of audio stream.
`sr`	Number of samples per second of audio stream (alias for *sampling_rate)*.
`sw`	Number of bytes used to represent one audio sample (alias for *sample_width*).

17.5 auditok.io.PyAudioSource

class `auditok.io.`**`PyAudioSource`**(*sampling_rate=16000*, *sample_width=2*, *channels=1*, *frames_per_buffer=1024*, *input_device_index=None*)

A class for an *AudioSource* that reads data from built-in microphone using PyAudio (https://people.csail.mit.edu/hubert/pyaudio/).

Parameters

- **`sampling_rate`** (*`int, default: 16000`*) – number of samples per second of audio data.
- **`sample_width`** (*`int, default: 2`*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **`channels`** (*`int, default: 1`*) – number of channels of audio data.
- **`frames_per_buffer`** (*`int, default: 1024`*) – PyAudio number of frames per buffer.
- **`input_device_index`** (*`None or int, default: None`*) – PyAudio index of audio device to read audio data from. If None default device is used.

`__init__`(*sampling_rate=16000, sample_width=2, channels=1, frames_per_buffer=1024, input_device_index=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([sampling_rate, sample_width, . . .])	Initialize self.
`close`()	Close audio source.
`is_open`()	Return True if audio source is open, False otherwise.
`open`()	Open audio source.
`read`(size)	Read and return *size* audio samples at most.

Attributes

`ch`	Number of channels in audio stream (alias for *channels*).
`channels`	Number of channels in audio stream.
`sample_width`	Number of bytes used to represent one audio sample.
`sampling_rate`	Number of samples per second of audio stream.
`sr`	Number of samples per second of audio stream (alias for *sampling_rate*).
`sw`	Number of bytes used to represent one audio sample (alias for *sample_width*).

17.6 auditok.io.StdinAudioSource

class `auditok.io.`**`StdinAudioSource`**(*sampling_rate=16000, sample_width=2, channels=1*)
A class for an *AudioSource* that reads data from standard input.

Parameters

- **`sampling_rate`** (*`int, default: 16000`*) – number of samples per second of audio data.
- **`sample_width`** (*`int, default: 2`*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **`channels`** (*`int, default: 1`*) – number of channels of audio data.

`__init__`(*sampling_rate=16000, sample_width=2, channels=1*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([sampling_rate, sample_width, channels])	Initialize self.
`close`()	Close audio source.
`is_open`()	Return True if audio source is open, False otherwise.
`open`()	Open audio source.
`read`(size)	Read and return *size* audio samples at most.

Attributes

`ch`	Number of channels in audio stream (alias for *channels*).
`channels`	Number of channels in audio stream.
`sample_width`	Number of bytes used to represent one audio sample.
`sampling_rate`	Number of samples per second of audio stream.
`sr`	Number of samples per second of audio stream (alias for *sampling_rate)*.
`sw`	Number of bytes used to represent one audio sample (alias for *sample_width*).

17.7 auditok.io.PyAudioPlayer

class `auditok.io.`**`PyAudioPlayer`**(*sampling_rate=16000*, *sample_width=2*, *channels=1*)

A class for audio playback using Pyaudio (https://people.csail.mit.edu/hubert/pyaudio/).

Parameters

- **sampling_rate** (*int, default: 16000*) – number of samples per second of audio data.
- **sample_width** (*int, default: 2*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **channels** (*int, default: 1*) – number of channels of audio data.

`__init__`(*sampling_rate=16000*, *sample_width=2*, *channels=1*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([sampling_rate, sample_width, channels])	Initialize self.
`play`(data[, progress_bar])	
`stop`()	

17.8 auditok.io.from_file

`auditok.io.`**`from_file`**(*filename*, *audio_format=None*, *large_file=False*, ***kwargs*)

Read audio data from *filename* and return an *AudioSource* object. if *audio_format* is None, the appropriate

AudioSource class is guessed from file's extension. *filename* can be a compressed audio or video file. This will require installing *pydub* (https://github.com/jiaaro/pydub).

The normal behavior is to load all audio data to memory from which a `BufferAudioSource` object is created. This should be convenient most of the time unless audio file is very large. In that case, and in order to load audio data in lazy manner (i.e. read data from disk each time `AudioSource.read()` is called), *large_file* should be True.

Note that the current implementation supports only wave and raw formats for lazy audio loading.

If an audio format is *raw*, the following keyword arguments are required:

- *sampling_rate*, *sr*: int, sampling rate of audio data.
- *sample_width*, *sw*: int, size in bytes of one audio sample.
- *channels*, *ch*: int, number of channels of audio data.

See also:

`to_file()`

Parameters

- **`filename`** (`str`) – path to input audio or video file.
- **`audio_format`** (`str`) – audio format used to save data (e.g. raw, webm, wav, ogg).
- **`large_file`** (`bool, default: False`) – if True, audio won't fully be loaded to memory but only when a window is read from disk.

Other Parameters

- **sampling_rate, sr** (*int*) – sampling rate of audio data
- **sample_width** (*int*) – sample width (i.e. number of bytes used to represent one audio sample)
- **channels** (*int*) – number of channels of audio data

Returns **audio_source** – an `AudioSource` object that reads data from input file.

Return type *AudioSource*

Raises *AudioIOError* – raised if audio data cannot be read in the given format or if *format* is *raw* and one or more audio parameters are missing.

17.9 auditok.io.to_file

`auditok.io.`**`to_file`**(*data*, *file*, *audio_format=None*, ***kwargs*)

Writes audio data to file. If *audio_format* is *None*, output audio format will be guessed from extension. If *audio_format* is *None* and *file* comes without an extension then audio data will be written as a raw audio file.

Parameters

- **`data`** (`bytes-like`) – audio data to be written. Can be a *bytes*, *bytearray*, *memoryview*, *array* or *numpy.ndarray* object.
- **`file`** (`str`) – path to output audio file.
- **`audio_format`** (`str`) – audio format used to save data (e.g. raw, webm, wav, ogg)
- **`kwargs`** (`dict`) – If an audio format other than *raw* is used, the following keyword arguments are required:

  - *sampling_rate*, *sr*: int, sampling rate of audio data.
  - *sample_width*, *sw*: int, size in bytes of one audio sample.
  - *channels*, *ch*: int, number of channels of audio data.

Raises

- *AudioParameterError* if output format is different than raw and one or more
- audio parameters are missing. *AudioIOError* if audio data cannot be written
- in the desired format.

17.10 auditok.io.player_for

`auditok.io.`**`player_for`**(*source*)

Return an *AudioPlayer* compatible with *source* (i.e., has the same sampling rate, sample width and number of channels).

Parameters **`source`** (`AudioSource`) – An object that has *sampling_rate*, *sample_width* and *sample_width* attributes.

Returns **player** – An audio player that has the same sampling rate, sample width and number of channels as *source*.

Return type *PyAudioPlayer*

`class` `auditok.io.`**`AudioSource`**(*sampling_rate*, *sample_width*, *channels*)

Base class for audio source objects.

Subclasses should implement methods to open/close and audio stream and read the desired amount of audio samples.

Parameters

- **`sampling_rate`** (*`int`*) – number of samples per second of audio data.
- **`sample_width`** (*`int`*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **`channels`** (*`int`*) – number of channels of audio data.

`ch`

Number of channels in audio stream (alias for *channels*).

`channels`

Number of channels in audio stream.

`close`()

Close audio source.

`is_open`()

Return True if audio source is open, False otherwise.

`open`()

Open audio source.

`read`(*size*)

Read and return *size* audio samples at most.

Parameters **`size`** (*`int`*) – Number of samples to read.

Returns

data – Audio data as a bytes object of length *N* * *sample_width* * *channels* where *N* equals:

- *size* if *size* <= remaining samples
- remaining samples if *size* > remaining samples

Return type bytes

`sample_width`
Number of bytes used to represent one audio sample.

`sampling_rate`
Number of samples per second of audio stream.

`sr`
Number of samples per second of audio stream (alias for *sampling_rate)*.

`sw`
Number of bytes used to represent one audio sample (alias for *sample_width*).

`class` `auditok.io.`**`Rewindable`**(*sampling_rate*, *sample_width*, *channels*)
Base class for rewindable audio streams.

Subclasses should implement a method to return back to the start of an the stream (*rewind*), as well as a property getter/setter named *position* that reads/sets stream position expressed in number of samples.

`position`
Return stream position in number of samples.

`position_ms`
Return stream position in milliseconds.

`position_s`
Return stream position in seconds.

`rewind`()
Go back to the beginning of audio stream.

`class` `auditok.io.`**`BufferAudioSource`**(*data*, *sampling_rate=16000*, *sample_width=2*, *channels=1*)
An *AudioSource* that encapsulates and reads data from a memory buffer.

This class implements the *Rewindable* interface. :param data: audio data :type data: bytes :param sampling_rate: number of samples per second of audio data. :type sampling_rate: int, default: 16000 :param sample_width: size in bytes of one audio sample. Possible values: 1, 2 or 4. :type sample_width: int, default: 2 :param channels: number of channels of audio data. :type channels: int, default: 1

`close`()
Close audio source.

`data`
Get raw audio data as a *bytes* object.

`is_open`()
Return True if audio source is open, False otherwise.

`open`()
Open audio source.

`position`
Get stream position in number of samples

position_ms
Get stream position in milliseconds.

read(*size*)
Read and return *size* audio samples at most.

Parameters **size** (*int*) – Number of samples to read.

Returns

data – Audio data as a bytes object of length *N* * *sample_width* * *channels* where *N* equals:

- *size* if *size* <= remaining samples
- remaining samples if *size* > remaining samples

Return type bytes

rewind()
Go back to the beginning of audio stream.

class auditok.io.**RawAudioSource**(*file*, *sampling_rate*, *sample_width*, *channels*)
A class for an *AudioSource* that reads data from a raw (headerless) audio file.

This class should be used for large raw audio files to avoid loading the whole data to memory.

Parameters

- **filename** (*str*) – path to a raw audio file.
- **sampling_rate** (*int*) – Number of samples per second of audio data.
- **sample_width** (*int*) – Size in bytes of one audio sample. Possible values : 1, 2, 4.
- **channels** (*int*) – Number of channels of audio data.

open()
Open audio source.

class auditok.io.**WaveAudioSource**(*filename*)
A class for an *AudioSource* that reads data from a wave file.

This class should be used for large wave files to avoid loading the whole data to memory.

Parameters **filename** (*str*) – path to a valid wave file.

open()
Open audio source.

class auditok.io.**PyAudioSource**(*sampling_rate=16000*, *sample_width=2*, *channels=1*, *frames_per_buffer=1024*, *input_device_index=None*)
A class for an *AudioSource* that reads data from built-in microphone using PyAudio (https://people.csail.mit.edu/hubert/pyaudio/).

Parameters

- **sampling_rate** (*int, default: 16000*) – number of samples per second of audio data.
- **sample_width** (*int, default: 2*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **channels** (*int, default: 1*) – number of channels of audio data.
- **frames_per_buffer** (*int, default: 1024*) – PyAudio number of frames per buffer.

- **input_device_index** (*None or int, default: None*) – PyAudio index of audio device to read audio data from. If None default device is used.

close()
Close audio source.

is_open()
Return True if audio source is open, False otherwise.

open()
Open audio source.

read(*size*)
Read and return *size* audio samples at most.

Parameters **size** (*int*) – Number of samples to read.

Returns

data – Audio data as a bytes object of length *N* * *sample_width* * *channels* where *N* equals:

- *size* if *size* <= remaining samples
- remaining samples if *size* > remaining samples

Return type bytes

class auditok.io.**StdinAudioSource**(*sampling_rate=16000*, *sample_width=2*, *channels=1*)
A class for an *AudioSource* that reads data from standard input.

Parameters

- **sampling_rate** (*int, default: 16000*) – number of samples per second of audio data.
- **sample_width** (*int, default: 2*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **channels** (*int, default: 1*) – number of channels of audio data.

close()
Close audio source.

is_open()
Return True if audio source is open, False otherwise.

open()
Open audio source.

class auditok.io.**PyAudioPlayer**(*sampling_rate=16000*, *sample_width=2*, *channels=1*)
A class for audio playback using Pyaudio (https://people.csail.mit.edu/hubert/pyaudio/).

Parameters

- **sampling_rate** (*int, default: 16000*) – number of samples per second of audio data.
- **sample_width** (*int, default: 2*) – size in bytes of one audio sample. Possible values: 1, 2 or 4.
- **channels** (*int, default: 1*) – number of channels of audio data.

auditok.io.**from_file**(*filename*, *audio_format=None*, *large_file=False*, ***kwargs*)
Read audio data from *filename* and return an *AudioSource* object. if *audio_format* is None, the appropriate *AudioSource* class is guessed from file's extension. *filename* can be a compressed audio or video file. This will require installing *pydub* (https://github.com/jiaaro/pydub).

The normal behavior is to load all audio data to memory from which a `BufferAudioSource` object is created. This should be convenient most of the time unless audio file is very large. In that case, and in order to load audio data in lazy manner (i.e. read data from disk each time `AudioSource.read()` is called), *large_file* should be True.

Note that the current implementation supports only wave and raw formats for lazy audio loading.

If an audio format is *raw*, the following keyword arguments are required:

- *sampling_rate*, *sr*: int, sampling rate of audio data.
- *sample_width*, *sw*: int, size in bytes of one audio sample.
- *channels*, *ch*: int, number of channels of audio data.

See also:

`to_file()`

Parameters

- **`filename`** (*`str`*) – path to input audio or video file.
- **`audio_format`** (*`str`*) – audio format used to save data (e.g. raw, webm, wav, ogg).
- **`large_file`** (*`bool, default: False`*) – if True, audio won't fully be loaded to memory but only when a window is read from disk.

Other Parameters

- **sampling_rate, sr** (*int*) – sampling rate of audio data
- **sample_width** (*int*) – sample width (i.e. number of bytes used to represent one audio sample)
- **channels** (*int*) – number of channels of audio data

Returns **audio_source** – an `AudioSource` object that reads data from input file.

Return type *AudioSource*

Raises *AudioIOError* – raised if audio data cannot be read in the given format or if *format* is *raw* and one or more audio parameters are missing.

`auditok.io.`**`to_file`**(*data*, *file*, *audio_format=None*, ***kwargs*)

Writes audio data to file. If *audio_format* is *None*, output audio format will be guessed from extension. If *audio_format* is *None* and *file* comes without an extension then audio data will be written as a raw audio file.

Parameters

- **`data`** (*`bytes-like`*) – audio data to be written. Can be a *bytes*, *bytearray*, *memoryview*, *array* or *numpy.ndarray* object.
- **`file`** (*`str`*) – path to output audio file.
- **`audio_format`** (*`str`*) – audio format used to save data (e.g. raw, webm, wav, ogg)
- **`kwargs`** (*`dict`*) – If an audio format other than *raw* is used, the following keyword arguments are required:
  - *sampling_rate*, *sr*: int, sampling rate of audio data.
  - *sample_width*, *sw*: int, size in bytes of one audio sample.
  - *channels*, *ch*: int, number of channels of audio data.

Raises

- *AudioParameterError* if output format is different than raw and one or more
- audio parameters are missing. *AudioIOError* if audio data cannot be written
- in the desired format.

`auditok.io.`**`player_for`**(*source*)

Return an *AudioPlayer* compatible with *source* (i.e., has the same sampling rate, sample width and number of channels).

Parameters **`source`** (`AudioSource`) – An object that has *sampling_rate*, *sample_width* and *sample_width* attributes.

Returns **player** – An audio player that has the same sampling rate, sample width and number of channels as *source*.

Return type *PyAudioPlayer*

CHAPTER 18

Signal processing

Module for basic audio signal processing and array operations.

to_array(data, sample_width, channels)	Extract individual channels of audio data and return a list of arrays of numeric samples.
extract_single_channel(data, fmt, channels, . . .)	
compute_average_channel(data, fmt, channels)	Compute and return average channel of multi-channel audio data.
compute_average_channel_stereo(data, . . .)	Compute and return average channel of stereo audio data.
separate_channels(data, fmt, channels)	Create a list of arrays of audio samples (*array.array* objects), one for each channel.
calculate_energy_single_channel(data, . . .)	Calculate the energy of mono audio data.
calculate_energy_multichannel(x, sample_width)	Calculate the energy of multi-channel audio data.

18.1 auditok.signal.to_array

auditok.signal.**to_array**(*data*, *sample_width*, *channels*)

Extract individual channels of audio data and return a list of arrays of numeric samples. This will always return a list of *array.array* objects (one per channel) even if audio data is mono.

Parameters

- **data** (*bytes*) – raw audio data.
- **sample_width** (*int*) – size in bytes of one audio sample (one channel considered).

Returns **samples_arrays** – list of arrays of audio samples.

Return type list

18.2 auditok.signal.extract_single_channel

`auditok.signal.`**`extract_single_channel`**(*data*, *fmt*, *channels*, *selected*)

18.3 auditok.signal.compute_average_channel

`auditok.signal.`**`compute_average_channel`**(*data*, *fmt*, *channels*)

Compute and return average channel of multi-channel audio data. If the number of channels is 2, use `compute_average_channel_stereo()` (much faster). This function uses satandard *array* module to convert *bytes* data into an array of numeric values.

Parameters

- **`data`** (*`bytes`*) – multi-channel audio data to mix down.
- **`fmt`** (*`str`*) – format (single character) to pass to *array.array* to convert *data* into an array of samples. This should be “b” if audio data’s sample width is 1, “h” if it’s 2 and “i” if it’s 4.
- **`channels`** (*`int`*) – number of channels of audio data.

Returns **mono_audio** – mixed down audio data.

Return type bytes

18.4 auditok.signal.compute_average_channel_stereo

`auditok.signal.`**`compute_average_channel_stereo`**(*data*, *sample_width*)

Compute and return average channel of stereo audio data. This function should be used when the number of channels is exactly 2 because in that case we can use standard *audioop* module which *much* faster then calling `compute_average_channel()`.

Parameters

- **`data`** (*`bytes`*) – 2-channel audio data to mix down.
- **`sample_width`** (*`int`*) – size in bytes of one audio sample (one channel considered).

Returns **mono_audio** – mixed down audio data.

Return type bytes

18.5 auditok.signal.separate_channels

`auditok.signal.`**`separate_channels`**(*data*, *fmt*, *channels*)

Create a list of arrays of audio samples (*array.array* objects), one for each channel.

Parameters

- **`data`** (*`bytes`*) – multi-channel audio data to mix down.
- **`fmt`** (*`str`*) – format (single character) to pass to *array.array* to convert *data* into an array of samples. This should be “b” if audio data’s sample width is 1, “h” if it’s 2 and “i” if it’s 4.
- **`channels`** (*`int`*) – number of channels of audio data.

Returns **channels_arr** – list of audio channels, each as a standard *array.array*.

Return type list

18.6 auditok.signal.calculate_energy_single_channel

`auditok.signal.`**`calculate_energy_single_channel`**`(`*data*, *sample_width*`)`

Calculate the energy of mono audio data. Energy is computed as:

$$energy = 20\log(\sqrt(1/N\sum_{i}^{N}{a_i}^2))wherea\_iisthei - thaudiosampleandNisthenumberofaudiosamplesindata.$$

Parameters

- **`data`** (*`bytes`*) – single-channel audio data.
- **`sample_width`** (*`int`*) – size in bytes of one audio sample.

Returns

energy – energy of audio signal.

Return type

float

18.7 auditok.signal.calculate_energy_multichannel

`auditok.signal.`**`calculate_energy_multichannel`**`(`*x*, *sample_width*, *aggregation_fn=<built-in function max>*`)`

Calculate the energy of multi-channel audio data. Energy is calculated channel-wise. An aggregation function is applied to the resulting energies (default: *max*). Also see `calculate_energy_single_channel()`.

Parameters

- **`data`** (*`bytes`*) – single-channel audio data.
- **`sample_width`** (*`int`*) – size in bytes of one audio sample (one channel considered).
- **`aggregation_fn`** (*`callable, default: max`*) – aggregation function to apply to the resulting per-channel energies.

Returns **energy** – aggregated energy of multi-channel audio signal.

Return type float

`auditok.signal.`**`calculate_energy_multichannel`**`(`*x*, *sample_width*, *aggregation_fn=<built-in function max>*`)`

Calculate the energy of multi-channel audio data. Energy is calculated channel-wise. An aggregation function is applied to the resulting energies (default: *max*). Also see `calculate_energy_single_channel()`.

Parameters

- **`data`** (*`bytes`*) – single-channel audio data.
- **`sample_width`** (*`int`*) – size in bytes of one audio sample (one channel considered).
- **`aggregation_fn`** (*`callable, default: max`*) – aggregation function to apply to the resulting per-channel energies.

Returns **energy** – aggregated energy of multi-channel audio signal.

Return type float

auditok.signal.**calculate_energy_single_channel**(*data*, *sample_width*)

Calculate the energy of mono audio data. Energy is computed as:

$$energy = 20\log(\sqrt(1/N\sum_{i}^{N} {a_i}^2)) where a\_i is the i-th audio sample and N is the number of audio samples in data.$$

Parameters

- **data** (*bytes*) – single-channel audio data.
- **sample_width** (*int*) – size in bytes of one audio sample.

Returns
energy – energy of audio signal.

Return type
float

auditok.signal.**compute_average_channel**(*data*, *fmt*, *channels*)

Compute and return average channel of multi-channel audio data. If the number of channels is 2, use *compute_average_channel_stereo()* (much faster). This function uses satandard *array* module to convert *bytes* data into an array of numeric values.

Parameters

- **data** (*bytes*) – multi-channel audio data to mix down.
- **fmt** (*str*) – format (single character) to pass to *array.array* to convert *data* into an array of samples. This should be "b" if audio data's sample width is 1, "h" if it's 2 and "i" if it's 4.
- **channels** (*int*) – number of channels of audio data.

Returns **mono_audio** – mixed down audio data.

Return type bytes

auditok.signal.**compute_average_channel_stereo**(*data*, *sample_width*)

Compute and return average channel of stereo audio data. This function should be used when the number of channels is exactly 2 because in that case we can use standard *audioop* module which *much* faster then calling *compute_average_channel()*.

Parameters

- **data** (*bytes*) – 2-channel audio data to mix down.
- **sample_width** (*int*) – size in bytes of one audio sample (one channel considered).

Returns **mono_audio** – mixed down audio data.

Return type bytes

auditok.signal.**separate_channels**(*data*, *fmt*, *channels*)

Create a list of arrays of audio samples (*array.array* objects), one for each channel.

Parameters

- **data** (*bytes*) – multi-channel audio data to mix down.
- **fmt** (*str*) – format (single character) to pass to *array.array* to convert *data* into an array of samples. This should be "b" if audio data's sample width is 1, "h" if it's 2 and "i" if it's 4.
- **channels** (*int*) – number of channels of audio data.

Returns **channels_arr** – list of audio channels, each as a standard *array.array*.

Return type list

`auditok.signal.`**`to_array`**(*data*, *sample_width*, *channels*)

Extract individual channels of audio data and return a list of arrays of numeric samples. This will always return a list of *array.array* objects (one per channel) even if audio data is mono.

Parameters

- **`data`** (*`bytes`*) – raw audio data.
- **`sample_width`** (*`int`*) – size in bytes of one audio sample (one channel considered).

Returns **samples_arrays** – list of arrays of audio samples.

Return type list

CHAPTER 19

Dataset

This module contains links to audio files that can be used for test purposes.

one_to_six_arabic_16000_mono_bc_noise	A wave file that contains a pronunciation of Arabic numbers from 1 to 6
was_der_mensch_saet_mono_44100_lead_trail_silence	A wave file that contains a sentence with a long leading and trailing silence

19.1 auditok.dataset.one_to_six_arabic_16000_mono_bc_noise

auditok.dataset.**one_to_six_arabic_16000_mono_bc_noise = '/home/docs/checkouts/readthedocs.**

A wave file that contains a pronunciation of Arabic numbers from 1 to 6

19.2 auditok.dataset.was_der_mensch_saet_mono_44100_lead_trail_silence

auditok.dataset.**was_der_mensch_saet_mono_44100_lead_trail_silence = '/home/docs/checkouts/r**

A wave file that contains a sentence with a long leading and trailing silence

auditok.dataset.**one_to_six_arabic_16000_mono_bc_noise = '/home/docs/checkouts/readthedocs.**

A wave file that contains a pronunciation of Arabic numbers from 1 to 6

auditok.dataset.**was_der_mensch_saet_mono_44100_lead_trail_silence = '/home/docs/checkouts/r**

A wave file that contains a sentence with a long leading and trailing silence

CHAPTER 20

License

MIT.

Python Module Index

a

auditok.core, 33
auditok.dataset, 79
auditok.io, 59
auditok.signal, 73
auditok.util, 47

Index

Symbols

__init__() (*auditok.core.AudioRegion method*), 36
__init__() (*auditok.core.StreamTokenizer method*), 39
__init__() (*auditok.io.AudioSource method*), 60
__init__() (*auditok.io.BufferAudioSource method*), 61
__init__() (*auditok.io.PyAudioPlayer method*), 64
__init__() (*auditok.io.PyAudioSource method*), 63
__init__() (*auditok.io.Rewindable method*), 60
__init__() (*auditok.io.StdinAudioSource method*), 63
__init__() (*auditok.io.WaveAudioSource method*), 62
__init__() (*auditok.util.AudioEnergyValidator method*), 48
__init__() (*auditok.util.AudioReader method*), 49
__init__() (*auditok.util.Recorder method*), 50

A

ads() (*auditok.util.ADSFactory static method*), 54
ADSFactory (*class in auditok.util*), 53
AudioDataSource (*in module auditok.util*), 55
AudioEnergyValidator (*class in auditok.util*), 47, 56
AudioReader (*class in auditok.util*), 48, 55
AudioRegion (*class in auditok.core*), 36, 41
AudioSource (*class in auditok.io*), 59, 66
auditok.core (*module*), 33
auditok.dataset (*module*), 79
auditok.io (*module*), 59
auditok.signal (*module*), 73
auditok.util (*module*), 47

B

BufferAudioSource (*class in auditok.io*), 61, 67

C

calculate_energy_multichannel() (*in module auditok.signal*), 75
calculate_energy_single_channel() (*in module auditok.signal*), 75
ch (*auditok.core.AudioRegion attribute*), 42
ch (*auditok.io.AudioSource attribute*), 66
channels (*auditok.core.AudioRegion attribute*), 42
channels (*auditok.io.AudioSource attribute*), 66
close() (*auditok.io.AudioSource method*), 66
close() (*auditok.io.BufferAudioSource method*), 67
close() (*auditok.io.PyAudioSource method*), 69
close() (*auditok.io.StdinAudioSource method*), 69
compute_average_channel() (*in module auditok.signal*), 74, 76
compute_average_channel_stereo() (*in module auditok.signal*), 74, 76

D

data (*auditok.io.BufferAudioSource attribute*), 67
DataSource (*class in auditok.util*), 53
DataValidator (*class in auditok.util*), 53
duration (*auditok.core.AudioRegion attribute*), 42

E

extract_single_channel() (*in module auditok.signal*), 74

F

from_file() (*in module auditok.io*), 64, 69

I

is_open() (*auditok.io.AudioSource method*), 66
is_open() (*auditok.io.BufferAudioSource method*), 67
is_open() (*auditok.io.PyAudioSource method*), 69
is_open() (*auditok.io.StdinAudioSource method*), 69
is_valid() (*auditok.util.AudioEnergyValidator method*), 57
is_valid() (*auditok.util.DataValidator method*), 53

L

len (*auditok.core.AudioRegion attribute*), 42

load() (*auditok.core.AudioRegion class method*), 42
load() (*in module auditok.core*), 33, 39

M

make_channel_selector() (*in module auditok.util*), 51, 52
make_duration_formatter() (*in module auditok.util*), 50, 52
millis (*auditok.core.AudioRegion attribute*), 42

O

one_to_six_arabic_16000_mono_bc_noise (*in module auditok.dataset*), 79
open() (*auditok.io.AudioSource method*), 66
open() (*auditok.io.BufferAudioSource method*), 67
open() (*auditok.io.PyAudioSource method*), 69
open() (*auditok.io.RawAudioSource method*), 68
open() (*auditok.io.StdinAudioSource method*), 69
open() (*auditok.io.WaveAudioSource method*), 68

P

play() (*auditok.core.AudioRegion method*), 42
player_for() (*in module auditok.io*), 66, 71
plot() (*auditok.core.AudioRegion method*), 42
position (*auditok.io.BufferAudioSource attribute*), 67
position (*auditok.io.Rewindable attribute*), 67
position_ms (*auditok.io.BufferAudioSource attribute*), 67
position_ms (*auditok.io.Rewindable attribute*), 67
position_s (*auditok.io.Rewindable attribute*), 67
PyAudioPlayer (*class in auditok.io*), 64, 69
PyAudioSource (*class in auditok.io*), 62, 68

R

RawAudioSource (*class in auditok.io*), 68
read() (*auditok.io.AudioSource method*), 66
read() (*auditok.io.BufferAudioSource method*), 68
read() (*auditok.io.PyAudioSource method*), 69
read() (*auditok.util.AudioReader method*), 56
read() (*auditok.util.DataSource method*), 53
read() (*auditok.util.StringDataSource method*), 53
Recorder (*class in auditok.util*), 50, 56
rewind() (*auditok.io.BufferAudioSource method*), 68
rewind() (*auditok.io.Rewindable method*), 67
Rewindable (*class in auditok.io*), 60, 67

S

sample_width (*auditok.core.AudioRegion attribute*), 43
sample_width (*auditok.io.AudioSource attribute*), 67
samples (*auditok.core.AudioRegion attribute*), 43
sampling_rate (*auditok.core.AudioRegion attribute*), 43
sampling_rate (*auditok.io.AudioSource attribute*), 67
save() (*auditok.core.AudioRegion method*), 43
seconds (*auditok.core.AudioRegion attribute*), 43
separate_channels() (*in module auditok.signal*), 74, 76
set_data() (*auditok.util.StringDataSource method*), 53
split() (*auditok.core.AudioRegion method*), 43
split() (*in module auditok.core*), 34, 40
split_and_plot() (*auditok.core.AudioRegion method*), 44
sr (*auditok.core.AudioRegion attribute*), 44
sr (*auditok.io.AudioSource attribute*), 67
StdinAudioSource (*class in auditok.io*), 63, 69
StreamTokenizer (*class in auditok.core*), 37, 44
StringDataSource (*class in auditok.util*), 53
sw (*auditok.core.AudioRegion attribute*), 44
sw (*auditok.io.AudioSource attribute*), 67

T

to_array() (*in module auditok.signal*), 73, 77
to_file() (*in module auditok.io*), 65, 70
tokenize() (*auditok.core.StreamTokenizer method*), 46

W

was_der_mensch_saet_mono_44100_lead_trail_silence (*in module auditok.dataset*), 79
WaveAudioSource (*class in auditok.io*), 62, 68