auditok Documentation

Release 0.1.5

Amine Sehili

Aug 24, 2020

Contents

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 and offers an easy to use API.

The latest version of this documentation can be found at Readthedocs.

CHAPTER 1

Requirements

auditok can be used with standard Python!

However, 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 back detections.
- matplotlib : plot audio signal and detections (see figures above).
- numpy : required by matplotlib. Also used for math operations instead of standard python if available.
- Optionally, you can use *sox* or *[p]arecord* for data acquisition and feed *auditok* using a pipe.

CHAPTER 2

Installation

Install with pip:

```
sudo pip install auditok
```

or install the latest version on Github:

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

CHAPTER 3

Getting started

3.1 *auditok* Command-line Usage Guide

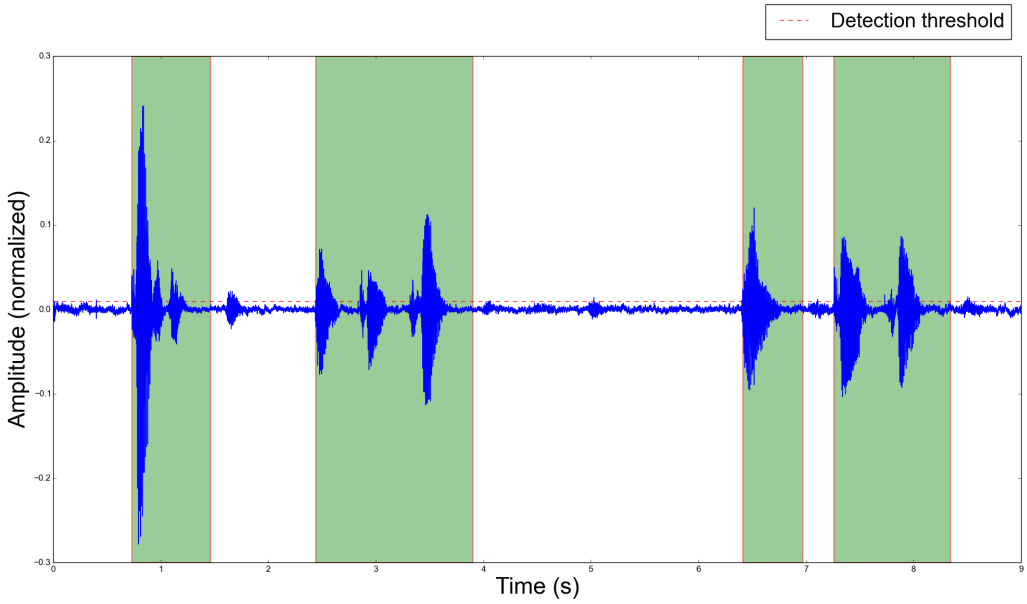
This user guide will go through a few of the most useful operations you can use **auditok** for and present two practical use cases.

Contents

3.1.1 Two-figure explanation

The following two figures illustrate an audio signal (blue) and regions detected as valid audio activities (green rectangles) according to a given threshold (red dashed line). They respectively depict the detection result when:

1. the detector tolerates phases of silence of up to 0.3 second (300 ms) within an audio activity (also referred to as acoustic event):



2. the detector splits an audio activity event into many activities if the within activity silence is over 0.2 second:

Beyond plotting signal and detections, you can play back audio activities as they are detected, save them or run a user command each time there is an activity, using, optionally, the file name of audio activity as an argument for the command.

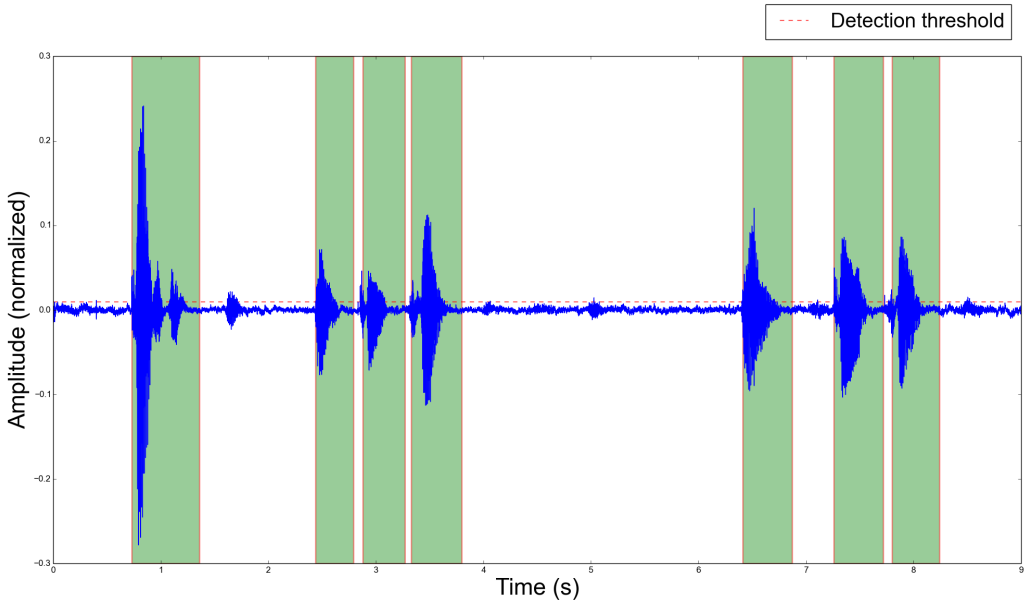
3.1.2 Command line usage

Try the detector with your voice

The first thing you want to check is perhaps how **auditok** detects your voice. If you have installed *PyAudio* just run (*Ctrl-C* to stop):

```
auditok
```

This will print **id start-time** and **end-time** for each detected activity. If you don't have *PyAudio*, you can use *sox* for data acquisition (*sudo apt-get install sox*) and tell **auditok** to read data from standard input:



rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok -i - -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	None	always signed integer

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

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

Play back detections

```
auditok -E
```

or

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

Option *-E* stands for echo, so **auditok** will play back whatever it detects. Using *-E* requires *PyAudio*, if you don't have *PyAudio* and want to play detections with sox, use the *-C* option:

```
rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok -i - -C "play -q -t raw -r␣
↪16000 -c 1 -b 16 -e signed $"
```

The *-C* option tells **auditok** to interpret its content as a command that should be run whenever **auditok** detects an audio activity, replacing the *$* by a name of a temporary file into which the activity is saved as raw audio. Here we use *play* to play the activity, giving the necessary *play* arguments for raw data.

rec and *play* are just an alias for *sox*.

The *-C* option can be useful in many cases. Imagine a command that sends audio data over a network only if there is an audio activity and saves bandwidth during silence.

Set detection threshold

If you notice that there are too many detections, use a higher value for energy threshold (the current version only implements a *validator* based on energy threshold. The use of spectral information is also desirable and might be part of future releases). To change the energy threshold (default: 50), use option *-e*:

```
auditok -E -e 55
```

or

```
rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok -i - -e 55 -C "play -q -t raw␣
↪-r 16000 -c 1 -b 16 -e signed $"
```

If however you figure out that the detector is missing some of or all your audio activities, use a lower value for *-e*.

Set format for printed detections information

By default, **auditok** prints the **id**, **start-time** and **end-time** of each detected activity:

```
1 1.87 2.67
2 3.05 3.73
3 3.97 4.49
...
```

If you want to customize the output format, use *–printf* option:

```
auditok -e 55 --printf "[{id}]: {start} to {end}"
```

output

```
[1]: 0.22 to 0.67
[2]: 2.81 to 4.18
[3]: 5.53 to 6.44
[4]: 7.32 to 7.82
...
```

Keywords *{id}*, *{start}* and *{end}* can be placed and repeated anywhere in the text. Time is shown in seconds, if you want a more detailed time information, use *–time-format*:

auditok -e 55 –printf “[{id}]: {start} to {end}” –time-format “%h:%m:%s.%i”

output

```
[1]: 00:00:01.080 to 00:00:01.760
[2]: 00:00:02.420 to 00:00:03.440
[3]: 00:00:04.930 to 00:00:05.570
[4]: 00:00:05.690 to 00:00:06.020
[5]: 00:00:07.470 to 00:00:07.980
...
```

Valid time directives are: *%h* (hours) *%m* (minutes) *%s* (seconds) *%i* (milliseconds). Two other directives, *%S* (default) and *%I* can be used for absolute time in seconds and milliseconds respectively.

1st Practical use case example: generate a subtitles template

Using *–printf ' and '–time-format*, the following command, used with an input audio or video file, will generate and an **srt** file template that can be later edited with a subtitles editor in a way that reduces the time needed to define when each utterance starts and where it ends:

```
auditok -e 55 -i input.wav -m 10 --printf "{id}\n{start} --> {end}\nPut some text␣
↪here...\n" --time-format "%h:%m:%s.%i"
```

output

```
1
00:00:00.730 --> 00:00:01.460
Put some text here...

2
00:00:02.440 --> 00:00:03.900
Put some text here...

3
00:00:06.410 --> 00:00:06.970
Put some text here...

4
00:00:07.260 --> 00:00:08.340
Put some text here...

5
00:00:09.510 --> 00:00:09.820
Put some text here...
```

2nd Practical use case example: build a (very) basic voice control application

This repository supplies a bash script the can send audio data to Google's Speech Recognition service and get its transcription. In the following we will use **auditok** as a lower layer component of a voice control application. The basic idea is to tell **auditok** to run, for each detected audio activity, a certain number of commands that make up the rest of our voice control application.

Assume you have installed **sox** and downloaded the Speech Recognition script. The sequence of commands to run is:

1- Convert raw audio data to flac using **sox**:

```
sox -t raw -r 16000 -c 1 -b 16 -e signed raw_input output.flac
```

2- Send flac audio data to Google and get its filtered transcription using speech-rec.sh :

```
speech-rec.sh -i output.flac -r 16000
```

3- Use **grep** to select lines that contain *transcript*:

```
grep transcript
```

4- Launch the following script, giving it the transcription as input:

```
#!/bin/bash
```

(continues on next page)

(continued from previous page)

```
read line

RES=`echo "$line" | grep -i "open firefox"`

if [[ $RES ]]
   then
     echo "Launch command: 'firefox &' ... "
     firefox &
     exit 0
fi

exit 0
```

As you can see, the script can handle one single voice command. It runs firefox if the text it receives contains **open firefox**. Save a script into a file named voice-control.sh (don't forget to run a **chmod u+x voice-control.sh**).

Now, thanks to option *-C*, we will use the four instructions with a pipe and tell **auditok** to run them each time it detects an audio activity. Try the following command and say *open firefox*:

```
rec -q -t raw -r 16000 -c 1 -b 16 -e signed - | auditok -M 5 -m 3 -n 1 --debug-file␣
↪file.log -e 60 -C "sox -t raw -r 16000 -c 1 -b 16 -e signed $ audio.flac ; speech-
↪rec.sh -i audio.flac -r 16000 | grep transcript | ./voice-control.sh"
```

Here we used option *-M 5* to limit the amount of read audio data to 5 seconds (**auditok** stops if there are no more data) and option *-n 1* to tell **auditok** to only accept tokens of 1 second or more and throw any token shorter than 1 second.

With *–debug-file file.log*, all processing steps are written into file.log with their timestamps, including any run command and the file name the command was given.

Plot signal and detections

use option *-p*. Requires *matplotlib* and *numpy*.

```
auditok ...  -p
```

Save plot as image or PDF

```
auditok ...  --save-image output.png
```

Requires *matplotlib* and *numpy*. Accepted formats: eps, jpeg, jpg, pdf, pgf, png, ps, raw, rgba, svg, svgz, tif, tiff.

Read data from file

```
auditok -i input.wav ...
```

Install *pydub* for other audio formats.

Limit the length of acquired data

```
auditok -M 12 ...
```

Time is in seconds. This is valid for data read from an audio device, stdin or an audio file.

Save the whole acquired audio signal

```
auditok -O output.wav ...
```

Install *pydub* for other audio formats.

Save each detection into a separate audio file

```
auditok -o det_{N}_{start}_{end}.wav ...
```

You can use a free text and place *{N}*, *{start}* and *{end}* wherever you want, they will be replaced by detection number, start time and end time respectively. Another example:

```
auditok -o {start}-{end}.wav ...
```

Install *pydub* for more audio formats.

Setting detection parameters

Alongside the threshold option *-e* seen so far, a couple of other options can have a great impact on the detector behavior. These options are summarized in the following table:

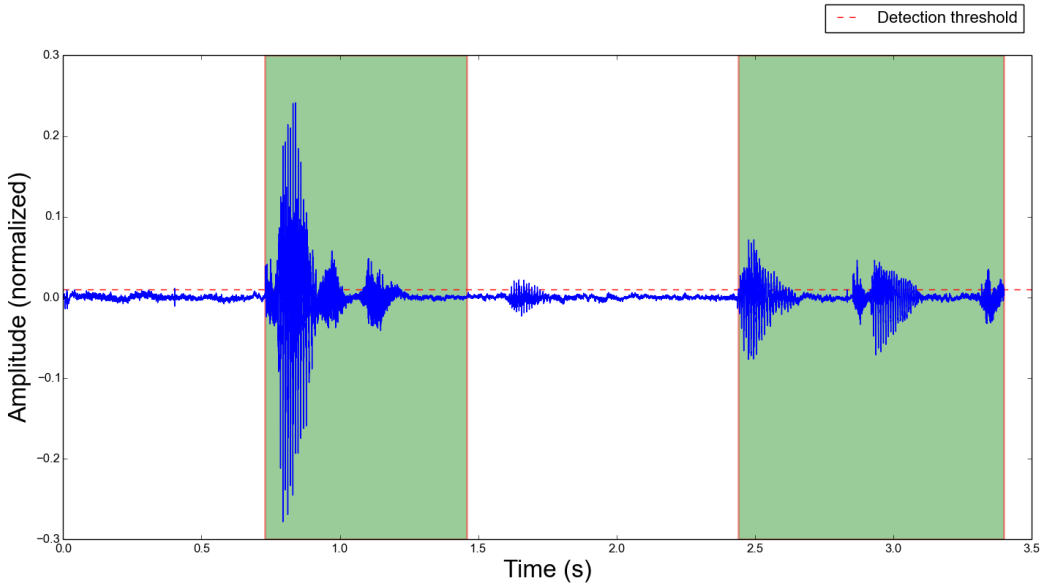
Option	Description	Unit	Default
-n	Minimum length an accepted audio activity should have	second	0.2 (200 ms)
-m	Maximum length an accepted audio activity should reach	second	5.
-s	Maximum length of a continuous silence period within an accepted audio activity	second	0.3 (300 ms)
-d	Drop trailing silence from an accepted audio activity	boolean	False
-a	Analysis window length (default value should be good)	second	0.01 (10 ms)

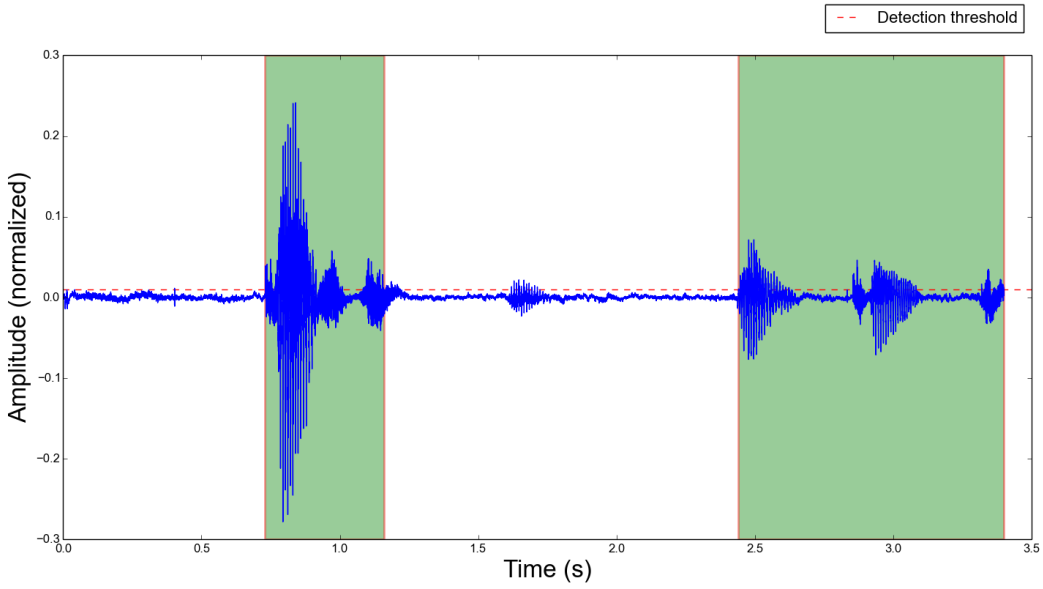
Normally, *auditok* does keeps trailing silence of a detected activity. Trailing silence is at most as long as maximum length of a continuous silence (option *-m*) and can be important for some applications such as speech recognition. If you want to drop trailing silence anyway use option *-d*. The following two figures show the output of the detector when it keeps the trailing silence and when it drops it respectively:

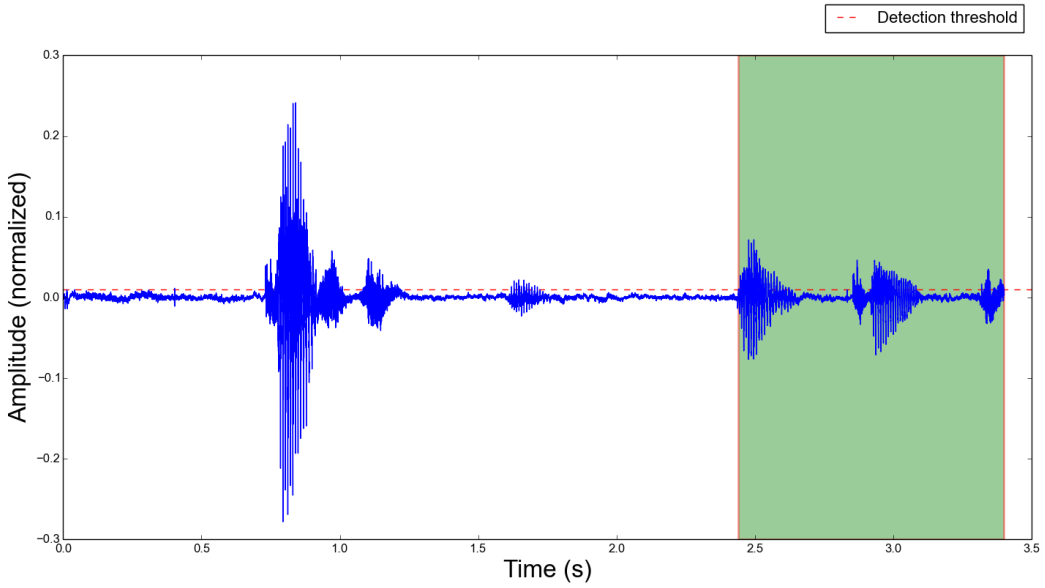
```
auditok ...  -d
```

You might want to only consider audio activities if they are above a certain duration. The next figure is the result of a detector that only accepts detections of 0.8 second and longer:

```
auditok ...  -n 0.8
```

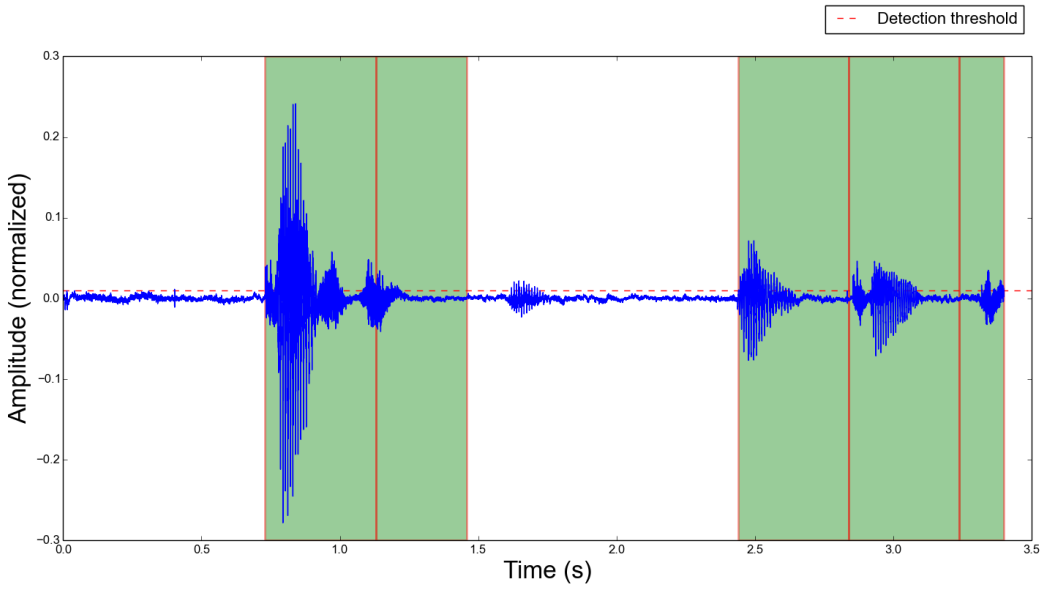

Detection threshold
Amplitude (normalized)
Time (s)
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Detection threshold
Amplitude (normalized)
Time (s)
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5



Finally it is almost always interesting to limit the length of detected audio activities. In any case, one does not want a too long audio event such as an alarm or a drill to hog the detector. For illustration purposes, we set the maximum duration to 0.4 second for this detector, so an audio activity is delivered as soon as it reaches 0.4 second:

```
auditok ...  -m 0.4
```



Debugging

If you want to print what happens when something is detected, use option *-D*.

```
auditok ...  -D
```

If you want to save everything into a log file, use *–debug-file file.log*.

```
auditok ...  --debug-file file.log
```

3.1.3 License

auditok is published under the GNU General Public License Version 3.

3.1.4 Author

Amine Sehili (<amine.sehili@gmail.com>)

3.2 *auditok* API Tutorial

Contents

auditok is a module that can be used as a generic tool for data tokenization. Although its core motivation is **Acoustic Activity Detection** (AAD) and extraction from audio streams (i.e. detect where a noise/an acoustic activity occurs within an audio stream and extract the corresponding portion of signal), it can easily be adapted to other tasks.

Globally speaking, it can be used to extract, from a sequence of observations, all sub-sequences that meet a certain number of criteria in terms of:

1. Minimum length of a **valid** token (i.e. sub-sequence)
2. Maximum length of a **valid** token
3. Maximum tolerated consecutive **non-valid** observations within a valid token

Examples of a non-valid observation are: a non-numeric ascii symbol if you are interested in sub-sequences of numeric symbols, or a silent audio window (of 10, 20 or 100 milliseconds for instance) if what interests you are audio regions made up of a sequence of "noisy" windows (whatever kind of noise: speech, baby cry, laughter, etc.).

The most important component of *auditok* is the `auditok.core.StreamTokenizer` class. An instance of this class encapsulates a `auditok.util.DataValidator` and can be configured to detect the desired regions from a stream. The `auditok.core.StreamTokenizer.tokenize()` method accepts a `auditok.util.DataSource` object that has a *read* method. Read data can be of any type accepted by the *validator*.

As the main aim of this module is **Audio Activity Detection**, it provides the `auditok.util.ADSFactory` factory class that makes it very easy to create an `auditok.util.ADSFactory.AudioDataSource` (a class that implements `auditok.util.DataSource`) object, be that from:

- A file on the disk
- A buffer of data
- The built-in microphone (requires PyAudio)

The `auditok.util.ADSFactory.AudioDataSource` class inherits from `auditok.util.DataSource` and supplies a higher abstraction level than `auditok.io.AudioSource` thanks to a bunch of handy features:

- Define a fixed-length *block_size* (alias *bs*, i.e. analysis window)
- Alternatively, use *block_dur* (duration in seconds, alias *bd*)
- Allow overlap between two consecutive analysis windows (if one of *hop_size* , *hs* or *hop_dur* , *hd* keywords is used and is > 0 and < *block_size*). This can be very important if your validator use the **spectral** information of audio data instead of raw audio samples.
- Limit the amount (i.e. duration) of read data (if keyword *max_time* or *mt* is used, very useful when reading data from the microphone)
- Record all read data and rewind if necessary (if keyword *record* or *rec* , also useful if you read data from the microphone and you want to process it many times off-line and/or save it)

See `auditok.util.ADSFactory` documentation for more information.

Last but not least, the current version has only one audio window validator based on signal energy (:class:‘auditok.util.AudioEnergyValidator).

3.2.1 Illustrative examples with strings

Let us look at some examples using the `auditok.util.StringDataSource` class created for test and illustration purposes. Imagine that each character of `auditok.util.StringDataSource` data represents an audio slice of 100 ms for example. In the following examples we will use upper case letters to represent noisy audio slices (i.e. analysis windows or frames) and lower case letter for silent frames.

Extract sub-sequences of consecutive upper case letters

We want to extract sub-sequences of characters that have:

- A minimum length of 1 (*min_length* = 1)
- A maximum length of 9999 (*max_length* = 9999)
- Zero consecutive lower case characters within them (*max_continuous_silence* = 0)

We also create the *UpperCaseChecker* with a *read* method that returns *True* if the checked character is in upper case and *False* otherwise.

```
from auditok import StreamTokenizer, StringDataSource, DataValidator

class UpperCaseChecker(DataValidator):
```

(continues on next page)

(continued from previous page)

```
   def is_valid(self, frame):
      return frame.isupper()

dsource = StringDataSource("aaaABCDEFbbGHIJKccc")
tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
             min_length=1, max_length=9999, max_continuous_silence=0)

tokenizer.tokenize(dsource)
```

The output is a list of two tuples, each contains the extracted sub-sequence and its start and end position in the original sequence respectively:

```
[(['A', 'B', 'C', 'D', 'E', 'F'], 3, 8), (['G', 'H', 'I', 'J', 'K'], 11, 15)]
```

Tolerate up to two non-valid (lower case) letters within an extracted sequence

To do so, we set *max_continuous_silence* =2:

```
from auditok import StreamTokenizer, StringDataSource, DataValidator

class UpperCaseChecker(DataValidator):
   def is_valid(self, frame):
      return frame.isupper()

dsource = StringDataSource("aaaABCDbbEFcGHIdddJKee")
tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
             min_length=1, max_length=9999, max_continuous_silence=2)

tokenizer.tokenize(dsource)
```

output:

```
[(['A', 'B', 'C', 'D', 'b', 'b', 'E', 'F', 'c', 'G', 'H', 'I', 'd', 'd'], 3, 16), (['J
↪', 'K', 'e', 'e'], 18, 21)]
```

Notice the trailing lower case letters "dd" and "ee" at the end of the two tokens. The default behavior of *auditok.core.StreamTokenizer* is to keep the *trailing silence* if it does not exceed *max_continuous_silence*. This can be changed using the *StreamTokenizer.DROP_TRAILING_SILENCE* mode (see next example).

Remove trailing silence

Trailing silence can be useful for many sound recognition applications, including speech recognition. Moreover, from the human auditory system point of view, trailing low energy signal helps removing abrupt signal cuts.

If you want to remove it anyway, you can do it by setting *mode* to *StreamTokenizer.DROP_TRAILING_SILENCE*:

```
from auditok import StreamTokenizer, StringDataSource, DataValidator

class UpperCaseChecker(DataValidator):
   def is_valid(self, frame):
      return frame.isupper()

dsource = StringDataSource("aaaABCDbbEFcGHIdddJKee")
tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
```

(continues on next page)

(continued from previous page)

```
            min_length=1, max_length=9999, max_continuous_silence=2,
            mode=StreamTokenizer.DROP_TRAILING_SILENCE)

tokenizer.tokenize(dsource)
```

output:

```
[(['A', 'B', 'C', 'D', 'b', 'b', 'E', 'F', 'c', 'G', 'H', 'I'], 3, 14), (['J', 'K'],␣
↪18, 19)]
```

Limit the length of detected tokens

Imagine that you just want to detect and recognize a small part of a long acoustic event (e.g. engine noise, water flow, etc.) and avoid that that event hogs the tokenizer and prevent it from feeding the event to the next processing step (i.e. a sound recognizer). You can do this by:

- limiting the length of a detected token.

and

- using a callback function as an argument to *auditok.core.StreamTokenizer.tokenize* so that the tokenizer delivers a token as soon as it is detected.

The following code limits the length of a token to 5:

```
from auditok import StreamTokenizer, StringDataSource, DataValidator

class UpperCaseChecker(DataValidator):
   def is_valid(self, frame):
      return frame.isupper()

dsource = StringDataSource("aaaABCDEFGHIJKbbb")
tokenizer = StreamTokenizer(validator=UpperCaseChecker(),
            min_length=1, max_length=5, max_continuous_silence=0)

def print_token(data, start, end):
    print("token = '{0}', starts at {1}, ends at {2}".format(''.join(data), start,␣
↪end))

tokenizer.tokenize(dsource, callback=print_token)
```

output:

```
"token = 'ABCDE', starts at 3, ends at 7"
"token = 'FGHIJ', starts at 8, ends at 12"
"token = 'K', starts at 13, ends at 13"
```

3.2.2 *auditok* and Audio Data

In the rest of this document we will use *auditok.util.ADSFactory*, *auditok.util.AudioEnergyValidator* and *auditok.core.StreamTokenizer* for Audio Activity Detection demos using audio data. Before we get any further it is worth, explaining a certain number of points.

auditok.util.ADSFactory.ads() method is used to create an *auditok.util.ADSFactory.AudioDataSource* object either from a wave file, the built-in microphone or a user-supplied data buffer. Refer to the API reference for more information and examples on `ADSFactory.ads()` and `AudioDataSource`.

The created `AudioDataSource` object is then passed to `StreamTokenizer.tokenize()` for tokenization.

auditok.util.ADSFactory.ads() accepts a number of keyword arguments, of which none is mandatory. The returned `AudioDataSource` object's features and behavior can however greatly differ depending on the passed arguments. Further details can be found in the respective method documentation.

Note however the following two calls that will create an `AudioDataSource` that reads data from an audio file and from the built-in microphone respectively.

```
from auditok import ADSFactory

# Get an AudioDataSource from a file
# use 'filename', alias 'fn' keyword argument
file_ads = ADSFactory.ads(filename = "path/to/file/")

# Get an AudioDataSource from the built-in microphone
# The returned object has the default values for sampling
# rate, sample width an number of channels. see method's
# documentation for customized values
mic_ads = ADSFactory.ads()
```

For `StreamTkenizer`, parameters *min_length*, *max_length* and *max_continuous_silence* are expressed in terms of number of frames. Each call to `AudioDataSource.read()` returns one frame of data or None.

If you want a *max_length* of 2 seconds for your detected sound events and your *analysis window* is *10 ms* long, you have to specify a *max_length* of 200 (*int(2. / (10. / 1000)) == 200*). For a *max_continuous_silence* of *300 ms* for instance, the value to pass to StreamTokenizer is 30 (*int(0.3 / (10. / 1000)) == 30*).

Each time `StreamTkenizer` calls the `read()` (has no argument) method of an `AudioDataSource` object, it returns the same amount of data, except if there are no more data (returns what's left in stream or None).

This fixed-length amount of data is referred here to as **analysis window** and is a parameter of `ADSFactory.ads()` method. By default `ADSFactory.ads()` uses an analysis window of 10 ms.

The number of samples that 10 ms of audio data contain will vary, depending on the sampling rate of your audio source/data (file, microphone, etc.). For a sampling rate of 16KHz (16000 samples per second), we have 160 samples for 10 ms.

You can use the *block_size* keyword (alias *bs*) to define your analysis window:

```
from auditok import ADSFactory

'''
Assume you have an audio file with a sampling rate of 16000
'''

# file_ads.read() will return blocks of 160 sample
file_ads = ADSFactory.ads(filename = "path/to/file/", block_size = 160)

# file_ads.read() will return blocks of 320 sample
file_ads = ADSFactory.ads(filename = "path/to/file/", bs = 320)
```

Fortunately, you can specify the size of your analysis window in seconds, thanks to keyword *block_dur* (alias *bd*):

```
from auditok import ADSFactory
# use an analysis window of 20 ms
file_ads = ADSFactory.ads(filename = "path/to/file/", bd = 0.02)
```

For `StreamTkenizer`, each `read()` call that does not return *None* is treated as a processing frame. `StreamTkenizer` has no way to figure out the temporal length of that frame (why sould it?). So to correctly initialize your `StreamTokenizer`, based on your analysis window duration, use something like:

```
analysis_win_seconds = 0.01 # 10 ms
my_ads = ADSFactory.ads(block_dur = analysis_win_seconds)
analysis_window_ms = analysis_win_seconds * 1000

# If you want your maximum continuous silence to be 300 ms use:
max_continuous_silence = int(300. / analysis_window_ms)

# which is the same as:
max_continuous_silence = int(0.3 / (analysis_window_ms / 1000))

# or simply:
max_continuous_silence = 30
```

3.2.3 Examples using real audio data

Extract isolated phrases from an utterance

We will build an *auditok.util.ADSFactory.AudioDataSource* using a wave file from the database. The file contains of isolated pronunciation of digits from 1 to 1 in Arabic as well as breath-in/out between 2 and 3. The code will play the original file then the detected sounds separately. Note that we use an *energy_threshold* of 65, this parameter should be carefully chosen. It depends on microphone quality, background noise and the amplitude of events you want to detect.

```
from auditok import ADSFactory, AudioEnergyValidator, StreamTokenizer, player_for,␣
↪dataset

# We set the `record` argument to True so that we can rewind the source
asource = ADSFactory.ads(filename=dataset.one_to_six_arabic_16000_mono_bc_noise,␣
↪record=True)

validator = AudioEnergyValidator(sample_width=asource.get_sample_width(), energy_
↪threshold=65)

# Default analysis window is 10 ms (float(asource.get_block_size()) / asource.get_
↪sampling_rate())
# min_length=20 : minimum length of a valid audio activity is 20 * 10 == 200 ms
# max_length=4000 :  maximum length of a valid audio activity is 400 * 10 == 4000 ms␣
↪== 4 seconds
# max_continuous_silence=30 : maximum length of a tolerated  silence within a valid␣
↪audio activity is 30 * 30 == 300 ms
tokenizer = StreamTokenizer(validator=validator, min_length=20, max_length=400, max_
↪continuous_silence=30)

asource.open()
tokens = tokenizer.tokenize(asource)

# Play detected regions back
```

(continues on next page)

(continued from previous page)

```
player = player_for(asource)

# Rewind and read the whole signal
asource.rewind()
original_signal = []

while True:
    w = asource.read()
    if w is None:
        break
    original_signal.append(w)

original_signal = ''.join(original_signal)

print("Playing the original file...")
player.play(original_signal)

print("playing detected regions...")
for t in tokens:
    print("Token starts at {0} and ends at {1}".format(t[1], t[2]))
    data = ''.join(t[0])
    player.play(data)

assert len(tokens) == 8
```

The tokenizer extracts 8 audio regions from the signal, including all isolated digits (from 1 to 6) as well as the 2-phase respiration of the subject. You might have noticed that, in the original file, the last three digit are closer to each other than the previous ones. If you wan them to be extracted as one single phrase, you can do so by tolerating a larger continuous silence within a detection:

```
tokenizer.max_continuous_silence = 50
asource.rewind()
tokens = tokenizer.tokenize(asource)

for t in tokens:
    print("Token starts at {0} and ends at {1}".format(t[1], t[2]))
    data = ''.join(t[0])
    player.play(data)

assert len(tokens) == 6
```

Trim leading and trailing silence

The tokenizer in the following example is set up to remove the silence that precedes the first acoustic activity or follows the last activity in a record. It preserves whatever it founds between the two activities. In other words, it removes the leading and trailing silence.

Sampling rate is 44100 sample per second, we'll use an analysis window of 100 ms (i.e. block_size == 4410)

Energy threshold is 50.

The tokenizer will start accumulating windows up from the moment it encounters the first analysis window of an energy >= 50. ALL the following windows will be kept regardless of their energy. At the end of the analysis, it will drop trailing windows with an energy below 50.

This is an interesting example because the audio file we're analyzing contains a very brief noise that occurs within the leading silence. We certainly do want our tokenizer to stop at this point and considers whatever it comes after as a useful signal. To force the tokenizer to ignore that brief event we use two other parameters *init_min* and *init_max_silence*. By *init_min* = 3 and *init_max_silence* = 1 we tell the tokenizer that a valid event must start with at least 3 noisy windows, between which there is at most 1 silent window.

Still with this configuration we can get the tokenizer detect that noise as a valid event (if it actually contains 3 consecutive noisy frames). To circumvent this we use an enough large analysis window (here of 100 ms) to ensure that the brief noise be surrounded by a much longer silence and hence the energy of the overall analysis window will be below 50.

When using a shorter analysis window (of 10 ms for instance, block_size == 441), the brief noise contributes more to energy calculation which yields an energy of over 50 for the window. Again we can deal with this situation by using a higher energy threshold (55 for example).

```
from auditok import ADSFactory, AudioEnergyValidator, StreamTokenizer, player_for,␣
↪dataset

# record = True so that we'll be able to rewind the source.
asource = ADSFactory.ads(filename=dataset.was_der_mensch_saet_mono_44100_lead_trail_
↪silence,
         record=True, block_size=4410)
asource.open()

original_signal = []
# Read the whole signal
while True:
   w = asource.read()
   if w is None:
      break
   original_signal.append(w)

original_signal = ''.join(original_signal)

# rewind source
asource.rewind()

# Create a validator with an energy threshold of 50
validator = AudioEnergyValidator(sample_width=asource.get_sample_width(), energy_
↪threshold=50)

# Create a tokenizer with an unlimited token length and continuous silence within a␣
↪token
# Note the DROP_TRAILING_SILENCE mode that will ensure removing trailing silence
trimmer = StreamTokenizer(validator, min_length = 20, max_length=99999999, init_min=3,
↪ init_max_silence=1, max_continuous_silence=9999999, mode=StreamTokenizer.DROP_
↪TRAILING_SILENCE)

tokens = trimmer.tokenize(asource)

# Make sure we only have one token
assert len(tokens) == 1, "Should have detected one single token"

trimmed_signal = ''.join(tokens[0][0])

player = player_for(asource)

print("Playing original signal (with leading and trailing silence)...")
```

(continues on next page)

(continued from previous page)

```
player.play(original_signal)
print("Playing trimmed signal...")
player.play(trimmed_signal)
```

Online audio signal processing

In the next example, audio data is directly acquired from the built-in microphone. The *auditok.core.StreamTokenizer.tokenize()* method is passed a callback function so that audio activities are delivered as soon as they are detected. Each detected activity is played back using the build-in audio output device.

As mentioned before , Signal energy is strongly related to many factors such microphone sensitivity, background noise (including noise inherent to the hardware), distance and your operating system sound settings. Try a lower *energy_threshold* if your noise does not seem to be detected and a higher threshold if you notice an over detection (echo method prints a detection where you have made no noise).

```
from auditok import ADSFactory, AudioEnergyValidator, StreamTokenizer, player_for

# record = True so that we'll be able to rewind the source.
# max_time = 10: read 10 seconds from the microphone
asource = ADSFactory.ads(record=True, max_time=10)

validator = AudioEnergyValidator(sample_width=asource.get_sample_width(), energy_
↪threshold=50)
tokenizer = StreamTokenizer(validator=validator, min_length=20, max_length=250, max_
↪continuous_silence=30)

player = player_for(asource)

def echo(data, start, end):
   print("Acoustic activity at: {0}--{1}".format(start, end))
   player.play(''.join(data))

asource.open()

tokenizer.tokenize(asource, callback=echo)
```

If you want to re-run the tokenizer after changing of one or many parameters, use the following code:

```
asource.rewind()
# change energy threshold for example
tokenizer.validator.set_energy_threshold(55)
tokenizer.tokenize(asource, callback=echo)
```

In case you want to play the whole recorded signal back use:

```
player.play(asource.get_audio_source().get_data_buffer())
```

3.2.4 Contributing

auditok is on GitHub. You're welcome to fork it and contribute.

Amine SEHILI <amine.sehili@gmail.com> September 2015

3.2.5 License

This package is published under GNU GPL Version 3.

CHAPTER 4

API Reference

4.1 auditok.core

This module gathers processing (i.e. tokenization) classes.

4.1.1 Class summary

StreamTokenizer(validator, min_length, ...)	Class for stream tokenizers.

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* :** instance of *DataValidator* that implements *is_valid* method.

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, default=0)*] 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, default=0)*] 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, default=0)*] *mode* can be:

1. *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* (given that the above conditions are fulfilled of course).

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 import StreamTokenizer, 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)
```

output

```
[(['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)
```

output

```
[(['A', 'A', 'A', 'A'], 3, 6)]
```

2. *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:

- a. *max_continuous_silence* is reached

or

- b. Its length reaches *max_length*. This is called 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 tailing silence can be removed if you use the *StreamTokenizer.DROP_TRAILING_SILENCE* mode.

Example

```
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)
```

output

```
[(['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)]
```

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

get_mode()

Return the current mode. To check whether a specific mode is activated use the bitwise 'and' operator *&*. Example:

```
if mode & self.STRICT_MIN_LENGTH != 0:
   do_something()
```

set_mode(*mode*)

Parameters

mode [*(int)*] New mode, must be one of:

- *StreamTokenizer.STRICT_MIN_LENGTH*
- *StreamTokenizer.DROP_TRAILING_SILENCE*
- *StreamTokenizer.STRICT_MIN_LENGTH* | *StreamTokenizer.DROP_TRAILING_SILENCE*
- *0*

See *StreamTokenizer.__init__* for more information about the mode.

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

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.

4.2 auditok.util

4.2.1 Class summary

DataSource	Base class for objects passed to *auditok.core.StreamTokenizer.tokenize()*.
StringDataSource(data)	A class that represent a *DataSource* as a string buffer.
ADSFactory	Factory class that makes it easy to create an *ADSFactory.AudioDataSource* object that implements *DataSource* and can therefore be passed to *auditok.core.StreamTokenizer.tokenize()*.
ADSFactory.AudioDataSource(audio_source, ...)	Base class for AudioDataSource objects.
ADSFactory.ADSDecorator(ads)	Base decorator class for AudioDataSource objects.
ADSFactory.OverlapADS(ads, hop_size)	A class for AudioDataSource objects that can read and return overlapping audio frames
ADSFactory.LimiterADS(ads, max_time)	A class for AudioDataSource objects that can read a fixed amount of data.
ADSFactory.RecorderADS(ads)	A class for AudioDataSource objects that can record all audio data they read, with a rewind facility.
DataValidator	Base class for a validator object used by *core.StreamTokenizer* to check if read data is valid.
AudioEnergyValidator(sample_width[, ...])	The most basic auditok audio frame validator.

class auditok.util.**DataSource**

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

read()

Read a piece 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*)
A 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 [] a basestring object.

read()
Read one character from buffer.

Returns Current character or None if end of buffer is reached

set_data(*data*)
Set a new data buffer.

Parameters

data [a basestring object] New data buffer.

class auditok.util.**ADSFactory**
Factory class that makes it easy to create an *ADSFactory.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 *ADSFactory.AudioDataSource* object.

There are many other features you want your *ADSFactory.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.

class ADSDecorator(*ads*)
Base decorator class for AudioDataSource objects.

class AudioDataSource(*audio_source*, *block_size*)
Base class for AudioDataSource objects. It inherits from DataSource and encapsulates an AudioSource object.

read()
Read a piece of data read from this source. If no more data is available, return None.

class LimiterADS(*ads*, *max_time*)
A class for AudioDataSource objects that can read a fixed amount of data. This can be useful when reading data from the microphone or from large audio files.

read()
Read a piece of data read from this source. If no more data is available, return None.

class OverlapADS(*ads*, *hop_size*)
A class for AudioDataSource objects that can read and return overlapping audio frames

read()
Read a piece of data read from this source. If no more data is available, return None.

class RecorderADS(*ads*)

A class for AudioDataSource objects that can record all audio data they read, with a rewind facility.

read()

Read a piece of data read from this source. If no more data is available, return None.

static ads(***kwargs*)

Create an return an *ADSFactory.AudioDataSource*. The type and behavior of the object is the result of the supplied parameters.

Parameters

No parameters [] read audio data from the available built-in microphone with the default parameters. The returned *ADSFactory.AudioDataSource* encapsulate an `io.PyAudioSource` object and hence it accepts the next four parameters are passed to use instead of their default values.

sampling_rate*, *sr [*(int)*] number of samples per second. Default = 16000.

sample_width*, *sw [*(int)*] number of bytes per sample (must be in (1, 2, 4)). Default = 2

channels*, *ch [*(int)*] number of audio channels. Default = 1 (only this value is currently accepted)

frames_per_buffer*, *fpb [*(int)*] number of samples of PyAudio buffer. Default = 1024.

audio_source*, *asrc [an *AudioSource* object] read data from this audio source

filename*, *fn [*(string)*] build an *io.AudioSource* object using this file (currently only wave format is supported)

data_buffer*, *db [*(string)*] 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.

max_time*, *mt [*(float)*] maximum time (in seconds) to read. Default behavior: read until there is no more data available.

record*, *rec [*(bool)*] save all read data in cache. Provide a navigable object which boasts a *rewind* method. Default = False.

block_dur*, *bd [*(float)*] 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)

hop_dur*, *hd [*(float)*] 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.

block_size*, *bs [*(int)*] 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.

hop_size*, *hs [*(int)*] 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

An AudioDataSource object that has the desired features.

Exampels

1. **Create an AudioDataSource that reads data from the microphone (requires Pyaudio) with default audio parameters:**

```
from auditok import ADSFactory
ads = ADSFactory.ads()
ads.get_sampling_rate()
16000
ads.get_sample_width()
2
ads.get_channels()
1
```

2. **Create an AudioDataSource that reads data from the microphone with a sampling rate of 48KHz:**

```
from auditok import ADSFactory
ads = ADSFactory.ads(sr=48000)
ads.get_sampling_rate()
48000
```

3. **Create an AudioDataSource that reads data from a wave file:**

```
import auditok
from auditok import ADSFactory
ads = ADSFactory.ads(fn=auditok.dataset.was_der_mensch_saet_mono_44100_lead_
↪trail_silence)
ads.get_sampling_rate()
44100
ads.get_sample_width()
2
ads.get_channels()
1
```

4. **Define size of read blocks as 20 ms**

```
import auditok
from auditok import ADSFactory
'''
we know samling rate for previous file is 44100 samples/second
so 10 ms are equivalent to 441 samples and 20 ms to 882
'''
block_size = 882
ads = ADSFactory.ads(bs = 882, fn=auditok.dataset.was_der_mensch_saet_mono_
↪44100_lead_trail_silence)
ads.open()
# read one block
data = ads.read()
ads.close()
len(data)
1764
assert len(data) ==  ads.get_sample_width() * block_size
```

5. **Define block size as a duration (use block_dur or bd):**

```
import auditok
from auditok import ADSFactory
dur = 0.25 # second
ads = ADSFactory.ads(bd = dur, fn=auditok.dataset.was_der_mensch_saet_mono_
↪44100_lead_trail_silence)
'''
we know samling rate for previous file is 44100 samples/second
for a block duration of 250 ms, block size should be 0.25 * 44100 = 11025
'''
ads.get_block_size()
11025
assert ads.get_block_size() ==  int(0.25 * 44100)
ads.open()
# read one block
data = ads.read()
ads.close()
len(data)
22050
assert len(data) ==  ads.get_sample_width() * ads.get_block_size()
```

6. **Read overlapping blocks (one of hope_size, hs, hop_dur or hd > 0):**

For better readability we'd better use *auditok.io.BufferAudioSource* with a string buffer:

```
import auditok
from auditok import ADSFactory
'''
we supply a data beffer instead of a file (keyword 'bata_buffer' or 'db')
sr : sampling rate = 16 samples/sec
sw : sample width = 1 byte
ch : channels = 1
'''
buffer = "abcdefghijklmnop" # 16 bytes = 1 second of data
bd = 0.250 # block duration = 250 ms = 4 bytes
hd = 0.125 # hop duration = 125 ms = 2 bytes
ads = ADSFactory.ads(db = "abcdefghijklmnop", bd = bd, hd = hd, sr = 16, sw =␣
↪1, ch = 1)
ads.open()
ads.read()
'abcd'
ads.read()
'cdef'
ads.read()
'efgh'
ads.read()
'ghij'
data = ads.read()
assert data == 'ijkl'
```

7. **Limit amount of read data (use max_time or mt):**

```
'''
We know audio file is larger than 2.25 seconds
We want to read up to 2.25 seconds of audio data
```

(continues on next page)

(continued from previous page)

```
'''
ads = ADSFactory.ads(mt = 2.25, fn=auditok.dataset.was_der_mensch_saet_mono_
↪44100_lead_trail_silence)
ads.open()
data = []
while True:
    d = ads.read()
    if d is None:
        break
    data.append(d)

ads.close()
data = b''.join(data)
assert len(data) == int(ads.get_sampling_rate() * 2.25 * ads.get_sample_
↪width() * ads.get_channels())
```

class auditok.util.**AudioEnergyValidator**(*sample_width*, *energy_threshold=45*)

The most basic auditok audio frame validator. This validator computes the log energy of an input audio frame and return True if the result is >= a given threshold, False otherwise.

Parameters

sample_width [*(int)*] Number of bytes of one audio sample. This is used to convert data from *basestring* or *Bytes* to an array of floats.

energy_threshold [*(float)*] A threshold used to check whether an input data buffer is valid.

is_valid(*data*)

Check if data is valid. Audio data will be converted into an array (of signed values) of which the log energy is computed. Log energy is computed as follows:

```
arr = AudioEnergyValidator._convert(signal, sample_width)
energy = float(numpy.dot(arr, arr)) / len(arr)
log_energy = 10. * numpy.log10(energy)
```

Parameters

data [either a *string* or a *Bytes* buffer] *data* is converted into a numerical array using the *sample_width* given in the constructor.

Returns

True if *log_energy* >= *energy_threshold*, False otherwise.

4.3 auditok.io

Module for low-level audio input-output operations.

4.3.1 Class summary

AudioSource([sampling_rate, sample_width, ...])	Base class for audio source objects.
Rewindable	Base class for rewindable audio streams.
BufferAudioSource(data_buffer[, ...])	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 the built-in microphone using 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

4.3.2 Function summary

from_file(filename)	Create an *AudioSource* object using the audio file specified by *filename*.
player_for(audio_source)	Return a PyAudioPlayer that can play data from *audio_source*.

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

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 stream. Default = 16000.

sample_width [int] Size in bytes of one audio sample. Possible values : 1, 2, 4. Default = 2.

channels [int] Number of channels of audio stream. The current version supports only mono audio streams (i.e. one channel).

ch

Return the number of channels of this audio source

channels

Number of channels of this audio source

close()

Close audio source

get_channels()

Return the number of channels of this audio source

get_sample_width()

Return the number of bytes used to represent one audio sample

get_sampling_rate()

Return the number of samples per second of audio stream

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] the number of samples to read.

Returns Audio data as a string of length 'N' * 'sample_width' * 'channels', where 'N' is:

- *size* if *size* < 'left_samples'
- 'left_samples' if *size* > 'left_samples'

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

sw
Number of bytes used to represent one audio sample

class auditok.io.**Rewindable**
Base class for rewindable audio streams. Subclasses should implement methods to return to the beginning of an audio stream as well as method to move to an absolute audio position expressed in time or in number of samples.

get_position()
Return the total number of already read samples

get_time_position()
Return the total duration in seconds of already read data

rewind()
Go back to the beginning of audio stream

set_position(*position*)
Move to an absolute position

Parameters

position [int] number of samples to skip from the start of the stream

set_time_position(*time_position*)
Move to an absolute position expressed in seconds

Parameters

time_position [float] seconds to skip from the start of the stream

class auditok.io.**BufferAudioSource**(*data_buffer*, *sampling_rate=16000*, *sample_width=2*, *channels=1*)
An *AudioSource* that encapsulates and reads data from a memory buffer. It implements methods from *Rewindable* and is therefore a navigable *AudioSource*.

append_data(*data_buffer*)
Append data to this audio stream

Parameters

data_buffer [str, basestring, Bytes] a buffer with a length multiple of (sample_width * channels)

close()
Close audio source

get_data_buffer()
Return all audio data as one string buffer.

get_position()
Return the total number of already read samples

get_time_position()
Return the total duration in seconds of already read data

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] the number of samples to read.

Returns Audio data as a string of length 'N' * 'sample_width' * 'channels', where 'N' is:

- *size* if *size* < 'left_samples'
- 'left_samples' if *size* > 'left_samples'

rewind()
Go back to the beginning of audio stream

set_data(*data_buffer*)
Set new data for this audio stream.

Parameters

data_buffer [str, basestring, Bytes] a string buffer with a length multiple of (sample_width * channels)

set_position(*position*)
Move to an absolute position

Parameters

position [int] number of samples to skip from the start of the stream

set_time_position(*time_position*)
Move to an absolute position expressed in seconds

Parameters

time_position [float] seconds to skip from the start of the stream

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

Parameters

filename **:** path to a valid wave file

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] the number of samples to read.

Returns Audio data as a string of length 'N' * 'sample_width' * 'channels', where 'N' is:

- *size* if *size* < 'left_samples'
- 'left_samples' if *size* > 'left_samples'

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 the built-in microphone using PyAudio.

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] the number of samples to read.

Returns Audio data as a string of length 'N' * 'sample_width' * 'channels', where 'N' is:

- *size* if *size* < 'left_samples'
- 'left_samples' if *size* > 'left_samples'

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

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] the number of samples to read.

Returns Audio data as a string of length 'N' * 'sample_width' * 'channels', where 'N' is:

- *size* if *size* < 'left_samples'

- 'left_samples' if *size* > 'left_samples'

class auditok.io.**PyAudioPlayer**(*sampling_rate=16000*, *sample_width=2*, *channels=1*)
A class for audio playback using Pyaudio

auditok.io.**from_file**(*filename*)
Create an *AudioSource* object using the audio file specified by *filename*. The appropriate *AudioSource* class is guessed from file's extension.

Parameters

***filename* :** path to an audio file.

Returns an *AudioSource* object that reads data from the given file.

auditok.io.**player_for**(*audio_source*)
Return a *PyAudioPlayer* that can play data from *audio_source*.

Parameters

audio_source [] an *AudioSource* object.

Returns *PyAudioPlayer* that has the same sampling rate, sample width and number of channels as *audio_source*.

4.4 auditok.dataset

This module contains links to audio files you can use for test purposes.

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 between long leading and trailing periods of silence

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

R

S

T

W