jiaaro / **pydub**

Branch: **master** **pydub** / **API.markdown** Find file Copy path

jiaaro Update API.markdown be1efc2 on Aug 2

7 contributors

570 lines (375 sloc) 22.2 KB

API Documentation

This document is a work in progress.

If you're looking for some functionality in particular, it's a good idea to take a look at the source code. Core functionality is mostly in `pydub/audio_segment.py` – a number of `AudioSegment` methods are in the `pydub/effects.py` module, and added to `AudioSegment` via the effect registration process (the `register_pydub_effect()` decorator function)

Currently Undocumented:

- Playback (`pydub.playback`)
- Signal Processing (compression, EQ, normalize, speed change - `pydub.effects` , `pydub.scipy_effects`)
- Signal generators (Sine, Square, Sawtooth, Whitenoise, etc - `pydub.generators`)
- Effect registration system (basically the `pydub.utils.register_pydub_effect` decorator)
- Silence utilities (detect silence, split on silence, etc - `pydub.silence`)

AudioSegment()

`AudioSegment` objects are immutable, and support a number of operators.

```
from pydub import AudioSegment
sound1 = AudioSegment.from_file("/path/to/sound.wav", format="wav")
sound2 = AudioSegment.from_file("/path/to/another_sound.wav", format="wav")

# sound1 6 dB louder, then 3.5 dB quieter
louder = sound1 + 6
quieter = sound1 - 3.5

# sound1, with sound2 appended
combined = sound1 + sound2

# sound1 repeated 3 times
repeated = sound1 * 3

# duration
duration_in_milliseconds = len(sound1)

# first 5 seconds of sound1
beginning = sound1[:5000]

# last 5 seconds of sound1
end = sound1[-5000:]

# Advanced usage, if you have raw audio data:
sound = AudioSegment(
    # raw audio data (bytes)
    data=b'…',
```

```
    # 2 byte (16 bit) samples
    sample_width=2,

    # 44.1 kHz frame rate
    frame_rate=44100,

    # stereo
    channels=2
)
```

Any operations that combine multiple `AudioSegment` objects in *any* way will first ensure that they have the same number of channels, frame rate, sample rate, bit depth, etc. When these things do not match, the lower quality sound is modified to match the quality of the higher quality sound so that quality is not lost: mono is converted to stereo, bit depth and frame rate/sample rate are increased as needed. If you do not want this behavior, you may explicitly reduce the number of channels, bits, etc using the appropriate `AudioSegment` methods.

AudioSegment(…).from_file()

Open an audio file as an `AudioSegment` instance and return it. there are also a number of wrappers provided for convenience, but you should probably just use this directly.

```
from pydub import AudioSegment

# wave and raw don't use ffmpeg
wav_audio = AudioSegment.from_file("/path/to/sound.wav", format="wav")
raw_audio = AudioSegment.from_file("/path/to/sound.raw", format="raw",
                                   frame_rate=44100, channels=2, sample_width=2)

# all other formats use ffmpeg
mp3_audio = AudioSegment.from_file("/path/to/sound.mp3", format="mp3")

# use a file you've already opened (advanced …ish)
with open("/path/to/sound.wav", "rb") as wav_file:
    audio_segment = AudioSegment.from_file(wav_file, format="wav")
```

The first argument is the path (as a string) of the file to read, **or** a file handle to read from.

Supported keyword arguments:

- `format` | example: `"aif"` | default: `"mp3"` Format of the output file. Supports `"wav"` and `"raw"` natively, requires ffmpeg for all other formats. `"raw"` files require 3 additional keyword arguments, `sample_width`, `frame_rate`, and `channels`, denoted below with: **`raw` only**. This extra info is required because raw audio files do not have headers to include this info in the file itself like wav files do.
- `sample_width` | example: `2` **`raw` only** — Use `1` for 8-bit audio `2` for 16-bit (CD quality) and `4` for 32-bit. It's the number of bytes per sample.
- `channels` | example: `1` **`raw` only** — `1` for mono, `2` for stereo.
- `frame_rate` | example: `2` **`raw` only** — Also known as sample rate, common values are `44100` (44.1kHz - CD audio), and `48000` (48kHz - DVD audio)

AudioSegment(…).export()

Write the `AudioSegment` object to a file – returns a file handle of the output file (you don't have to do anything with it, though).

```
from pydub import AudioSegment
sound = AudioSegment.from_file("/path/to/sound.wav", format="wav")

# simple export
file_handle = sound.export("/path/to/output.mp3", format="mp3")

# more complex export
```

```
file_handle = sound.export("/path/to/output.mp3",
                           format="mp3",
                           bitrate="192k",
                           tags={"album": "The Bends", "artist": "Radiohead"},
                           cover="/path/to/albumcovers/radioheadthebends.jpg")
```

The first argument is the location (as a string) to write the output, **or** a file handle to write to. If you do not pass an output file or path, a temporary file is generated.

Supported keyword arguments:

- `format` | example: `"aif"` | default: `"mp3"` Format of the output file. Supports `"wav"` and `"raw"` natively, requires ffmpeg for all other formats.
- `codec` | example: `"libvorbis"` For formats that may contain content encoded with different codecs, you can specify the codec you'd like the encoder to use. For example, the "ogg" format is often used with the "libvorbis" codec. (requires ffmpeg)
- `bitrate` | example: `"128k"` For compressed formats, you can pass the bitrate you'd like the encoder to use (requires ffmpeg). Each codec accepts different bitrate arguments so take a look at the ffmpeg documentation for details (bitrate usually shown as `-b`, `-ba` or `-a:b`).
- `tags` | example: `{"album": "1989", "artist": "Taylor Swift"}` Allows you to supply media info tags for the encoder (requires ffmpeg). Not all formats can receive tags (mp3 can).
- `parameters` | example: `["-ac", "2"]` Pass additional commpand line parameters to the ffmpeg call. These are added to the end of the call (in the output file section).
- `id3v2_version` | example: `"3"` | default: `"4"` Set the ID3v2 version used by ffmpeg to add tags to the output file. If you want Windows Exlorer to display tags, use `"3"` here (source).
- `cover` | example: `"/path/to/imgfile.png"` Allows you to supply a cover image (path to the image file). Currently, only MP3 files allow this keyword argument. Cover image must be a jpeg, png, bmp, or tiff file.

AudioSegment.empty()

Creates a zero-duration `AudioSegment`.

```
from pydub import AudioSegment
empty = AudioSegment.empty()

len(empty) == 0
```

This is useful for aggregation loops:

```
from pydub import AudioSegment

sounds = [
  AudioSegment.from_wav("sound1.wav"),
  AudioSegment.from_wav("sound2.wav"),
  AudioSegment.from_wav("sound3.wav"),
]

playlist = AudioSegment.empty()
for sound in sounds:
  playlist += sound
```

AudioSegment.silent()

Creates a silent audiosegment, which can be used as a placeholder, spacer, or as a canvas to overlay other sounds on top of.

```
from pydub import AudioSegment
```

```
ten_second_silence = AudioSegment.silent(duration=10000)
```

Supported keyword arguments:

- `duration` | example: `3000` | default: `1000` (1 second) Length of the silent `AudioSegment`, in milliseconds
- `frame_rate` | example `44100` | default: `11025` (11.025 kHz) Frame rate (i.e., sample rate) of the silent `AudioSegment` in Hz

AudioSegment.from_mono_audiosegments()

Creates a multi-channel audiosegment out of multiple mono audiosegments (two or more). Each mono audiosegment passed in should be exactly the same length, down to the frame count.

```
from pydub import AudioSegment

left_channel = AudioSegment.from_wav("sound1.wav")
right_channel = AudioSegment.from_wav("sound1.wav")

stereo_sound = AudioSegment.from_mono_audiosegments(left_channel, right_channel)
```

AudioSegment(…).dBFS

Returns the loudness of the `AudioSegment` in dBFS (db relative to the maximum possible loudness). A Square wave at maximum amplitude will be roughly 0 dBFS (maximum loudness), whereas a Sine Wave at maximum amplitude will be roughly -3 dBFS.

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

loudness = sound.dBFS
```

AudioSegment(…).channels

Number of channels in this audio segment (1 means mono, 2 means stereo)

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

channel_count = sound.channels
```

AudioSegment(…).sample_width

Number of bytes in each sample (1 means 8 bit, 2 means 16 bit, etc). CD Audio is 16 bit, (sample width of 2 bytes).

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

bytes_per_sample = sound.sample_width
```

AudioSegment(…).frame_rate

CD Audio has a 44.1kHz sample rate, which means `frame_rate` will be `44100` (same as sample rate, see `frame_width`). Common values are `44100` (CD), `48000` (DVD), `22050`, `24000`, `12000` and `11025`.

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

frames_per_second = sound.frame_rate
```

AudioSegment(…).frame_width

Number of bytes for each "frame". A frame contains a sample for each channel (so for stereo you have 2 samples per frame, which are played simultaneously). `frame_width` is equal to `channels * sample_width`. For CD Audio it'll be `4` (2 channels times 2 bytes per sample).

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

bytes_per_frame = sound.frame_width
```

AudioSegment(…).rms

A measure of loudness. Used to compute dBFS, which is what you should use in most cases. Loudness is logarithmic (rms is not), which makes dB a much more natural scale.

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

loudness = sound.rms
```

AudioSegment(…).max

The highest amplitude of any sample in the `AudioSegment`. Useful for things like normalization (which is provided in `pydub.effects.normalize`).

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

peak_amplitude = sound.max
```

AudioSegment(…).max_dBFS

The highest amplitude of any sample in the `AudioSegment`, in dBFS (relative to the highest possible amplitude value). Useful for things like normalization (which is provided in `pydub.effects.normalize`).

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

normalized_sound = sound.apply_gain(-sound.max_dBFS)
```

AudioSegment(…).duration_seconds

Returns the duration of the `AudioSegment` in seconds (`len(sound)` returns milliseconds). This is provided for convenience; it calls `len()` internally.

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

assert sound.duration_seconds == (len(sound) / 1000.0)
```

AudioSegment(…).raw_data

The raw audio data of the AudioSegment. Useful for interacting with other audio libraries or weird APIs that want audio data in the form of a bytestring. Also comes in handy if you're implementing effects or other direct signal processing.

You probably don't need this, but if you do… you'll know.

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

raw_audio_data = sound.raw_data
```

AudioSegment(…).frame_count()

Returns the number of frames in the `AudioSegment`. Optionally you may pass in a `ms` keywork argument to retrieve the number of frames in that number of milliseconds of audio in the `AudioSegment` (useful for slicing, etc).

```
from pydub import AudioSegment
sound = AudioSegment.from_file("sound1.wav")

number_of_frames_in_sound = sound.frame_count()

number_of_frames_in_200ms_of_sound = sound.frame_count(ms=200)
```

Supported keyword arguments:

- `ms` | example: `3000` | default: `None` (entire duration of `AudioSegment`) When specified, method returns number of frames in X milliseconds of the `AudioSegment`

AudioSegment(…).append()

Returns a new `AudioSegment`, created by appending another `AudioSegment` to this one (i.e., adding it to the end), Optionally using a crossfade. `AudioSegment(…).append()` is used internally when adding `AudioSegment` objects together with the `+` operator.

By default a 100ms (0.1 second) crossfade is used to eliminate pops and crackles.

```
from pydub import AudioSegment
sound1 = AudioSegment.from_file("sound1.wav")
sound2 = AudioSegment.from_file("sound2.wav")

# default 100 ms crossfade
combined = sound1.append(sound2)

# 5000 ms crossfade
combined_with_5_sec_crossfade = sound1.append(sound2, crossfade=5000)

# no crossfade
no_crossfade1 = sound1.append(sound2, crossfade=0)

# no crossfade
no_crossfade2 = sound1 + sound2
```

Supported keyword arguments:

- `crossfade` | example: `3000` | default: `100` (entire duration of `AudioSegment`) When specified, method returns number of frames in X milliseconds of the `AudioSegment`

AudioSegment(…).overlay()

Overlays an `AudioSegment` onto this one. In the resulting `AudioSegment` they will play simultaneously. If the overlaid `AudioSegment` is longer than this one, the result will be truncated (so the end of the overlaid sound will be cut off). The result is always the same length as this `AudioSegment` even when using the `loop`, and `times` keyword arguments.

Since `AudioSegment` objects are immutable, you can get around this by overlaying the shorter sound on the longer one, or by creating a silent `AudioSegment` with the appropriate duration, and overlaying both sounds on to that one.

```
from pydub import AudioSegment
sound1 = AudioSegment.from_file("sound1.wav")
sound2 = AudioSegment.from_file("sound2.wav")

played_togther = sound1.overlay(sound2)

sound2_starts_after_delay = sound1.overlay(sound2, position=5000)

volume_of_sound1_reduced_during_overlay = sound1.overlay(sound2, gain_during_overlay=-8)

sound2_repeats_until_sound1_ends = sound1.overlay(sound2, loop=true)

sound2_plays_twice = sound1.overlay(sound2, times=2)

# assume sound1 is 30 sec long and sound2 is 5 sec long:
sound2_plays_a_lot = sound1.overlay(sound2, times=10000)
len(sound1) == len(sound2_plays_a_lot)
```

Supported keyword arguments:

- `position` | example: `3000` | default: `0` (beginning of this `AudioSegment`) The overlaid `AudioSegment` will not begin until X milliseconds have passed
- `loop` | example: `True` | default: `False` (entire duration of `AudioSegment`) The overlaid `AudioSegment` will repeat (starting at `position`) until the end of this `AudioSegment`
- `times` | example: `4` | default: `1` (entire duration of `AudioSegment`) The overlaid `AudioSegment` will repeat X times (starting at `position`) but will still be truncated to the length of this `AudioSegment`
- `gain_during_overlay` | example: `-6.0` | default: `0` (no change in volume during overlay) Change the original audio by this many dB while overlaying audio. This can be used to make the original audio quieter while the overlaid audio plays.

AudioSegment(...).apply_gain(`gain`)

Change the amplitude (generally, loudness) of the `AudioSegment`. Gain is specified in dB. This method is used internally by the `+` operator.

```
from pydub import AudioSegment
sound1 = AudioSegment.from_file("sound1.wav")

# make sound1 louder by 3.5 dB
louder_via_method = sound1.apply_gain(+3.5)
louder_via_operator = sound1 + 3.5

# make sound1 quieter by 5.7 dB
quieter_via_method = sound1.apply_gain(-5.7)
quieter_via_operator = sound1 - 5.7
```

AudioSegment(...).fade()

A more general (more flexible) fade method. You may specify `start` and `end`, or one of the two along with duration (e.g., `start` and `duration`).

```
from pydub import AudioSegment
sound1 = AudioSegment.from_file("sound1.wav")

fade_louder_for_3_seconds_in_middle = sound1.fade(to_gain=+6.0, start=7500, duration=3000)

fade_quieter_beteen_2_and_3_seconds = sound1.fade(to_gain=-3.5, start=2000, end=3000)

# easy way is to use the .fade_in() convenience method. note: -120dB is basically silent.
fade_in_the_hard_way = sound1.fade(from_gain=-120.0, start=0, duration=5000)
fade_out_the_hard_way = sound1.fade(to_gain=-120.0, end=0, duration=5000)
```

Supported keyword arguments:

- `to_gain` | example: `-3.0` | default: `0` (0dB, no change) Resulting change at the end of the fade. `-6.0` means fade will be be from 0dB (no change) to -6dB, and everything after the fade will be -6dB.
- `from_gain` | example: `-3.0` | default: `0` (0dB, no change) Change at the beginning of the fade. `-6.0` means fade (and all audio before it) will be be at -6dB will fade up to 0dB – the rest of the audio after the fade will be at 0dB (i.e., unchanged).
- `start` | example: `7500` | NO DEFAULT Position to begin fading (in milliseconds). `5500` means fade will begin after 5.5 seconds.
- `end` | example: `4` | NO DEFAULT The overlaid `AudioSegment` will repeat X times (starting at `position`) but will still be truncated to the length of this `AudioSegment`
- `duration` | example: `4` | NO DEFAULT You can use `start` or `end` with duration, instead of specifying both - provided as a convenience.

AudioSegment(…).fade_out()

Fade out (to silent) the end of this `AudioSegment`. Uses `.fade()` internally.

Supported keyword arguments:

- `duration` | example: `5000` | NO DEFAULT How long (in milliseconds) the fade should last. Passed directly to `.fade()` internally

AudioSegment(…).fade_in()

Fade in (from silent) the beginning of this `AudioSegment`. Uses `.fade()` internally.

Supported keyword arguments:

- `duration` | example: `5000` | NO DEFAULT How long (in milliseconds) the fade should last. Passed directly to `.fade()` internally

AudioSegment(…).reverse()

Make a copy of this `AudioSegment` that plays backwards. Useful for Pink Floyd, screwing around, and some audio processing algorithms.

AudioSegment(…).set_sample_width()

Creates an equivalent version of this `AudioSegment` with the specified sample width (in bytes). Increasing this value does not generally cause a reduction in quality. Reducing it *definitely* does cause a loss in quality. Higher Sample width means more dynamic range.

AudioSegment(…).set_frame_rate()

Creates an equivalent version of this `AudioSegment` with the specified frame rate (in Hz). Increasing this value does not generally cause a reduction in quality. Reducing it *definitely does* cause a loss in quality. Higher frame rate means larger frequency response (higher frequencies can be represented).

AudioSegment(…).set_channels()

Creates an equivalent version of this `AudioSegment` with the specified number of channels (1 is Mono, 2 is Stereo). Converting from mono to stereo does not cause any audible change. Converting from stereo to mono may result in loss of quality (but only if the left and right chanels differ).

AudioSegment(…).split_to_mono()

Splits a stereo `AudioSegment` into two, one for each channel (Left/Right). Returns a list with the new `AudioSegment` objects with the left channel at index 0 and the right channel at index 1.

AudioSegment(…).apply_gain_stereo()

```
from pydub import AudioSegment
sound1 = AudioSegment.from_file("sound1.wav")

# make left channel 6dB quieter and right channe 2dB louder
stereo_balance_adjusted = sound1.apply_gain_stereo(-6, +2)
```

Apply gain to the left and right channel of a stereo `AudioSegment` . If the `AudioSegment` is mono, it will be converted to stereo before applying the gain.

Both gain arguments are specified in dB.

AudioSegment(…).pan()

```
from pydub import AudioSegment
sound1 = AudioSegment.from_file("sound1.wav")

# pan the sound 15% to the right
panned_right = sound1.pan(+0.15)

# pan the sound 50% to the left
panned_left = sound1.pan(-0.50)
```

Takes one positional argument, *pan amount*, which should be between -1.0 (100% left) and +1.0 (100% right)

When pan_amount == 0.0 the left/right balance is not changed.

Panning does not alter the *perceived* loundness, but since loudness is decreasing on one side, the other side needs to get louder to compensate. When panned hard left, the left channel will be 3dB louder and the right channel will be silent (and vice versa).

AudioSegment(…).get_array_of_samples()

Returns the raw audio data as an array of (numeric) samples. Note: if the audio has multiple channels, the samples for each channel will be serialized – for example, stereo audio would look like `[sample_1_L, sample_1_R, sample_2_L, sample_2_R, …]` .

This method is mainly for use in implementing effects, and other processing.

```
from pydub import AudioSegment
sound = AudioSegment.from_file(“sound1.wav”)

samples = sound.get_array_of_samples()

# then modify samples...

new_sound = sound._spawn(samples)
```

note that when using numpy or scipy you will need to convert back to an array before you spawn:

```
import array
import numpy as np
from pydub import AudioSegment

sound = AudioSegment.from_file(“sound1.wav”)
samples = sound.get_array_of_samples()

shifted_samples = np.right_shift(samples, 1)

# now you have to convert back to an array.array
shifted_samples_array = array.array(sound.array_type, shifted_samples)
```

```
new_sound = sound._spawn(shifted_samples_array)
```

AudioSegment(…).get_dc_offset()

Returns a value between -1.0 and 1.0 representing the DC offset of a channel. This is calculated using `audioop.avg()` and normalizing the result by samples max value.

Supported keyword arguments:

- `channel` | example: `2` | default: `1` Selects left (1) or right (2) channel to calculate DC offset. If segment is mono, this value is ignored.

AudioSegment(…).remove_dc_offset()

Removes DC offset from channel(s). This is done by using `audioop.bias()`, so watch out for overflows.

Supported keyword arguments:

- `channel` | example: `2` | default: None Selects left (1) or right (2) channel remove DC offset. If value if None, removes from all available channels. If segment is mono, this value is ignored.
- `offset` | example: `-0.1` | default: None Offset to be removed from channel(s). Calculates offset if it's None. Offset values must be between -1.0 and 1.0.

Effects

Collection of DSP effects that are implemented by `AudioSegment` objects.

AudioSegment(…).invert_phase()

Make a copy of this `AudioSegment` and inverts the phase of the signal. Can generate anti-phase waves for noise suppression or cancellation.