22.1. `audioop` — Manipulate raw audio data

The `audioop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16, 24 or 32 bits wide, stored in bytes-like objects. All scalar items are integers, unless specified otherwise.

Changed in version 3.4: Support for 24-bit samples was added. All functions now accept any bytes-like object. String input now results in an immediate error.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

exception `audioop.`**`error`**

    This exception is raised on all errors, such as unknown number of bytes per sample, etc.

`audioop.`**`add`**(*fragment1*, *fragment2*, *width*)

    Return a fragment which is the addition of the two samples passed as parameters. *width* is the sample width in bytes, either `1`, `2`, `3` or `4`. Both fragments should have the same length. Samples are truncated in case of overflow.

`audioop.`**`adpcm2lin`**(*adpcmfragment*, *width*, *state*)

    Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm()` for details on ADPCM coding. Return a tuple (`sample, newstate`) where the sample has the width specified in *width*.

`audioop.`**`alaw2lin`**(*fragment*, *width*)

    Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

`audioop.`**`avg`**(*fragment*, *width*)

    Return the average over all samples in the fragment.

`audioop.`**`avgpp`**(*fragment*, *width*)

    Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

audioop.**bias**(*fragment*, *width*, *bias*)

Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

audioop.**byteswap**(*fragment*, *width*)

"Byteswap" all samples in a fragment and returns the modified fragment. Converts big-endian samples to little-endian and vice versa.

New in version 3.4.

audioop.**cross**(*fragment*, *width*)

Return the number of zero crossings in the fragment passed as an argument.

audioop.**findfactor**(*fragment*, *reference*)

Return a factor *F* such that `rms(add(fragment, mul(reference, -F)))` is minimal, i.e., return the factor with which you should multiply *reference* to make it match as well as possible to *fragment*. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to `len(fragment)`.

audioop.**findfit**(*fragment*, *reference*)

Try to match *reference* as well as possible to a portion of *fragment* (which should be the longer fragment). This is (conceptually) done by taking slices out of *fragment*, using `findfactor()` to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple (`offset`, `factor`) where *offset* is the (integer) offset into *fragment* where the optimal match started and *factor* is the (floating-point) factor as per `findfactor ()`.

audioop.**findmax**(*fragment*, *length*)

Search *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e., return *i* for which `rms(fragment[i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to `len(fragment)`.

audioop.**getsample**(*fragment*, *width*, *index*)

Return the value of sample *index* from the fragment.

audioop.**lin2adpcm**(*fragment*, *width*, *state*)

Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM al-

gorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple (adpcmfrag, newstate), and the *newstate* should be passed to the next call of lin2adpcm(). In the initial call, None can be passed as the state. *adpcmfrag* is the ADPCM coded fragment packed 2 4-bit values per byte.

audioop.**lin2alaw**(*fragment*, *width*)

Convert samples in the audio fragment to a-LAW encoding and return this as a bytes object. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)

Convert samples between 1-, 2-, 3- and 4-byte formats.

> **Note:**   In some audio formats, such as .WAV files, 16, 24 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:
>
> ```
> new_frames = audioop.lin2lin(frames, old_width, 1)
> new_frames = audioop.bias(new_frames, 1, 128)
> ```
>
> The same, in reverse, has to be applied when converting from 8 to 16, 24 or 32 bit width samples.

audioop.**lin2ulaw**(*fragment*, *width*)

Convert samples in the audio fragment to u-LAW encoding and return this as a bytes object. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.**max**(*fragment*, *width*)

Return the maximum of the *absolute value* of all samples in a fragment.

audioop.**maxpp**(*fragment*, *width*)

Return the maximum peak-peak value in the sound fragment.

audioop.**minmax**(*fragment*, *width*)

Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

audioop.**mul**(*fragment*, *width*, *factor*)

Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Samples are truncated in case of overflow.

audioop. **ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])

Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple (`newfragment, newstate`), and *newstate* should be passed to the next call of `ratecv()`. The initial call should pass `None` as the state.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to `1` and `0` respectively.

audioop. **reverse**(*fragment*, *width*)

Reverse the samples in a fragment and returns the modified fragment.

audioop. **rms**(*fragment*, *width*)

Return the root-mean-square of the fragment, i.e. `sqrt(sum(S_i^2)/n)`.

This is a measure of the power in an audio signal.

audioop. **tomono**(*fragment*, *width*, *lfactor*, *rfactor*)

Convert a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *rfactor* before adding the two channels to give a mono signal.

audioop. **tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)

Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

audioop. **ulaw2lin**(*fragment*, *width*)

Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as `mul()` or `max()` make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```python
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
```

```
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the *initial* state (the one you passed to lin2adpcm()) along to the decoder, not the final state (as returned by the coder). If you want to use struct.Struct to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The find*() routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)]
    #                out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postf
    return audioop.add(inputdata, outputdata, 2)
```