CHAPTER

TWENTYTHREE

MULTIMEDIA SERVICES

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here's an overview:

23.1 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 algorithm 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:

```
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)
```

(continues on next page)

(continued from previous page)

```
    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) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

23.2 `aifc` — Read and write AIFF and AIFC files

Source code: Lib/aifc.py

This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of `nchannels * samplesize` bytes, and a second's worth of audio consists of `nchannels * samplesize * framerate` bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes (176,400 bytes).

Module *aifc* defines the following function:

`aifc.open`(*file*, *mode=None*)
: Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument *file* is either a string naming a file or a *file object*. *mode* must be `'r'` or `'rb'` when the

file must be opened for reading, or `'w'` or `'wb'` when the file must be opened for writing. If omitted, `file.mode` is used if it exists, otherwise `'rb'` is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`. The *open()* function may be used in a `with` statement. When the `with` block completes, the *close()* method is called.

Changed in version 3.4: Support for the `with` statement was added.

Objects returned by *open()* when a file is opened for reading have the following methods:

aifc.getnchannels()
Return the number of audio channels (1 for mono, 2 for stereo).

aifc.getsampwidth()
Return the size in bytes of individual samples.

aifc.getframerate()
Return the sampling rate (number of audio frames per second).

aifc.getnframes()
Return the number of audio frames in the file.

aifc.getcomptype()
Return a bytes array of length 4 describing the type of compression used in the audio file. For AIFF files, the returned value is `b'NONE'`.

aifc.getcompname()
Return a bytes array convertible to a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is `b'not compressed'`.

aifc.getparams()
Returns a *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

aifc.getmarkers()
Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

aifc.getmark(*id*)
Return the tuple as described in *getmarkers()* for the mark with the given *id*.

aifc.readframes(*nframes*)
Read and return the next *nframes* frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

aifc.rewind()
Rewind the read pointer. The next *readframes()* will start from the beginning.

aifc.setpos(*pos*)
Seek to the specified frame number.

aifc.tell()
Return the current frame number.

aifc.close()
Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by *open()* when a file is opened for writing have all the above methods, except for `readframes()` and `setpos()`. In addition the following methods exist. The `get*()` methods can only be called after the corresponding `set*()` methods have been called. Before the first `writeframes()` or `writeframesraw()`, all parameters except for the number of frames must be filled in.

aifc.aiff()
Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in `'.aiff'` in which case the default is an AIFF file.

aifc.aifc()
Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in `'.aiff'` in which case the default is an AIFF file.

aifc.setnchannels(*nchannels*)
Specify the number of channels in the audio file.

aifc.setsampwidth(*width*)
Specify the size in bytes of audio samples.

aifc.setframerate(*rate*)
Specify the sampling frequency in frames per second.

aifc.setnframes(*nframes*)
Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

aifc.setcomptype(*type*, *name*)
Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type as a bytes array, the type parameter should be a bytes array of length 4. Currently the following compression types are supported: `b'NONE'`, `b'ULAW'`, `b'ALAW'`, `b'G722'`.

aifc.setparams(*nchannels*, *sampwidth*, *framerate*, *comptype*, *compname*)
Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a *getparams()* call as argument to *setparams()*.

aifc.setmark(*id*, *pos*, *name*)
Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before *close()*.

aifc.tell()
Return the current write position in the output file. Useful in combination with *setmark()*.

aifc.writeframes(*data*)
Write data to the output file. This method can only be called after the audio file parameters have been set.

Changed in version 3.4: Any *bytes-like object* is now accepted.

aifc.writeframesraw(*data*)
Like *writeframes()*, except that the header of the audio file is not updated.

Changed in version 3.4: Any *bytes-like object* is now accepted.

aifc.close()
Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

23.3 sunau — Read and write Sun AU files

Source code: Lib/sunau.py

The *sunau* module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules *aifc* and *wave*.

An audio file consists of a header followed by the data. The fields of the header are:

Field	Contents
magic word	The four bytes `.snd`.
header size	Size of the header, including info, in bytes.
data size	Physical size of the data, in bytes.
encoding	Indicates how the audio samples are encoded.
sample rate	The sampling rate.
# of channels	The number of channels in the samples.
info	ASCII string giving a description of the audio file (padded with null bytes).

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The *sunau* module defines the following functions:

sunau.open(*file*, *mode*)
If *file* is a string, open the file by that name, otherwise treat it as a seekable file-like object. *mode* can be any of

'r' Read only mode.

'w' Write only mode.

Note that it does not allow read/write files.

A *mode* of **'r'** returns an **AU_read** object, while a *mode* of **'w'** or **'wb'** returns an **AU_write** object.

sunau.openfp(*file*, *mode*)
A synonym for *open()*, maintained for backwards compatibility.

Deprecated since version 3.7, will be removed in version 3.9.

The *sunau* module defines the following exception:

exception sunau.Error
An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The *sunau* module defines the following data items:

sunau.AUDIO_FILE_MAGIC
An integer every valid Sun AU file begins with, stored in big-endian form. This is the string `.snd` interpreted as an integer.

sunau.AUDIO_FILE_ENCODING_MULAW_8
sunau.AUDIO_FILE_ENCODING_LINEAR_8
sunau.AUDIO_FILE_ENCODING_LINEAR_16
sunau.AUDIO_FILE_ENCODING_LINEAR_24
sunau.AUDIO_FILE_ENCODING_LINEAR_32
sunau.AUDIO_FILE_ENCODING_ALAW_8
Values of the encoding field from the AU header which are supported by this module.

sunau.AUDIO_FILE_ENCODING_FLOAT
sunau.AUDIO_FILE_ENCODING_DOUBLE
sunau.AUDIO_FILE_ENCODING_ADPCM_G721
sunau.AUDIO_FILE_ENCODING_ADPCM_G722
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_3
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_5
Additional known values of the encoding field from the AU header, but which are not supported by this module.

23.3.1 AU_read Objects

AU_read objects, as returned by *open()* above, have the following methods:

AU_read.close()
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

AU_read.getnchannels()
Returns number of audio channels (1 for mono, 2 for stereo).

AU_read.getsampwidth()
Returns sample width in bytes.

AU_read.getframerate()
Returns sampling frequency.

AU_read.getnframes()
Returns number of audio frames.

AU_read.getcomptype()
Returns compression type. Supported compression types are `'ULAW'`, `'ALAW'` and `'NONE'`.

AU_read.getcompname()
Human-readable version of *getcomptype()*. The supported types have the respective names `'CCITT G.711 u-law'`, `'CCITT G.711 A-law'` and `'not compressed'`.

AU_read.getparams()
Returns a *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

AU_read.readframes(*n*)
Reads and returns at most *n* frames of audio, as a *bytes* object. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

AU_read.rewind()
Rewind the file pointer to the beginning of the audio stream.

The following two methods define a term "position" which is compatible between them, and is otherwise implementation dependent.

AU_read.setpos(*pos*)
Set the file pointer to the specified position. Only values returned from *tell()* should be used for *pos*.

AU_read.tell()
Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the *aifc*, and don't do anything interesting.

AU_read.getmarkers()
Returns `None`.

AU_read.getmark(*id*)
Raise an error.

23.3.2 AU_write Objects

AU_write objects, as returned by *open()* above, have the following methods:

AU_write.setnchannels(*n*)
Set the number of channels.

`AU_write.setsampwidth(`*n*`)`
Set the sample width (in bytes.)

Changed in version 3.4: Added support for 24-bit samples.

`AU_write.setframerate(`*n*`)`
Set the frame rate.

`AU_write.setnframes(`*n*`)`
Set the number of frames. This can be later changed, when and if more frames are written.

`AU_write.setcomptype(`*type*, *name*`)`
Set the compression type and description. Only `'NONE'` and `'ULAW'` are supported on output.

`AU_write.setparams(`*tuple*`)`
The *tuple* should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Set all parameters.

`AU_write.tell()`
Return current position in the file, with the same disclaimer for the *`AU_read.tell()`* and *`AU_read.setpos()`* methods.

`AU_write.writeframesraw(`*data*`)`
Write audio frames, without correcting *nframes*.

Changed in version 3.4: Any *bytes-like object* is now accepted.

`AU_write.writeframes(`*data*`)`
Write audio frames and make sure *nframes* is correct.

Changed in version 3.4: Any *bytes-like object* is now accepted.

`AU_write.close()`
Make sure *nframes* is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`.

23.4 `wave` — Read and write WAV files

Source code: Lib/wave.py

The *`wave`* module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The *`wave`* module defines the following function and exception:

`wave.open(`*file*, *mode=None*`)`
If *file* is a string, open the file by that name, otherwise treat it as a file-like object. *mode* can be:

`'rb'` Read only mode.

`'wb'` Write only mode.

Note that it does not allow read/write WAV files.

A *mode* of `'rb'` returns a `Wave_read` object, while a *mode* of `'wb'` returns a `Wave_write` object. If *mode* is omitted and a file-like object is passed as *file*, `file.mode` is used as the default value for *mode*.

If you pass in a file-like object, the wave object will not close it when its `close()` method is called; it is the caller's responsibility to close the file object.

The *open()* function may be used in a `with` statement. When the `with` block completes, the *Wave_read.close()* or *Wave_write.close()* method is called.

Changed in version 3.4: Added support for unseekable files.

`wave.openfp`(*file*, *mode*)
A synonym for *open()*, maintained for backwards compatibility.

Deprecated since version 3.7, will be removed in version 3.9.

exception `wave.Error`
An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

23.4.1 Wave_read Objects

Wave_read objects, as returned by *open()*, have the following methods:

`Wave_read.close()`
Close the stream if it was opened by *wave*, and make the instance unusable. This is called automatically on object collection.

`Wave_read.getnchannels()`
Returns number of audio channels (`1` for mono, `2` for stereo).

`Wave_read.getsampwidth()`
Returns sample width in bytes.

`Wave_read.getframerate()`
Returns sampling frequency.

`Wave_read.getnframes()`
Returns number of audio frames.

`Wave_read.getcomptype()`
Returns compression type (`'NONE'` is the only supported type).

`Wave_read.getcompname()`
Human-readable version of *getcomptype()*. Usually `'not compressed'` parallels `'NONE'`.

`Wave_read.getparams()`
Returns a *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

`Wave_read.readframes`(*n*)
Reads and returns at most *n* frames of audio, as a *bytes* object.

`Wave_read.rewind()`
Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the *aifc* module, and don't do anything interesting.

`Wave_read.getmarkers()`
Returns `None`.

`Wave_read.getmark`(*id*)
Raise an error.

The following two methods define a term "position" which is compatible between them, and is otherwise implementation dependent.

`Wave_read.setpos`(*pos*)
Set the file pointer to the specified position.

`Wave_read.tell()`
Return current file pointer position.

23.4.2 Wave_write Objects

For seekable output streams, the `wave` header will automatically be updated to reflect the number of frames actually written. For unseekable streams, the *nframes* value must be accurate when the first frame data is written. An accurate *nframes* value can be achieved either by calling *setnframes()* or *setparams()* with the number of frames that will be written before *close()* is called and then using *writeframesraw()* to write the frame data, or by calling *writeframes()* with all of the frame data to be written. In the latter case *writeframes()* will calculate the number of frames in the data and set *nframes* accordingly before writing the frame data.

Wave_write objects, as returned by *open()*, have the following methods:

Changed in version 3.4: Added support for unseekable files.

`Wave_write.close()`
Make sure *nframes* is correct, and close the file if it was opened by *wave*. This method is called upon object collection. It will raise an exception if the output stream is not seekable and *nframes* does not match the number of frames actually written.

`Wave_write.setnchannels(`*n*`)`
Set the number of channels.

`Wave_write.setsampwidth(`*n*`)`
Set the sample width to *n* bytes.

`Wave_write.setframerate(`*n*`)`
Set the frame rate to *n*.

Changed in version 3.2: A non-integral input to this method is rounded to the nearest integer.

`Wave_write.setnframes(`*n*`)`
Set the number of frames to *n*. This will be changed later if the number of frames actually written is different (this update attempt will raise an error if the output stream is not seekable).

`Wave_write.setcomptype(`*type*, *name*`)`
Set the compression type and description. At the moment, only compression type `NONE` is supported, meaning no compression.

`Wave_write.setparams(`*tuple*`)`
The *tuple* should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Sets all parameters.

`Wave_write.tell()`
Return current position in the file, with the same disclaimer for the *Wave_read.tell()* and *Wave_read.setpos()* methods.

`Wave_write.writeframesraw(`*data*`)`
Write audio frames, without correcting *nframes*.

Changed in version 3.4: Any *bytes-like object* is now accepted.

`Wave_write.writeframes(`*data*`)`
Write audio frames and make sure *nframes* is correct. It will raise an error if the output stream is not seekable and the total number of frames that have been written after *data* has been written does not match the previously set value for *nframes*.

Changed in version 3.4: Any *bytes-like object* is now accepted.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`, and any attempt to do so will raise *wave.Error*.

23.5 chunk — Read IFF chunked data

Source code: Lib/chunk.py

This module provides an interface for reading files that use EA IFF 85 chunks.[1] This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

Offset	Length	Contents
0	4	Chunk ID
4	4	Size of chunk in big-endian byte order, not including the header
8	*n*	Data bytes, where *n* is the size given in the preceding field
8 + *n*	0 or 1	Pad byte needed if *n* is odd and chunk alignment is used

The ID is a 4-byte string which identifies the type of chunk.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the *Chunk* class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with an *EOFError* exception.

class chunk.Chunk(*file*, *align=True*, *bigendian=True*, *inclheader=False*)

Class which represents a chunk. The *file* argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is `read()`. If the methods *seek()* and *tell()* are present and don't raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument *align* is true, chunks are assumed to be aligned on 2-byte boundaries. If *align* is false, no alignment is assumed. The default value is true. If the optional argument *bigendian* is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument *inclheader* is true, the size given in the chunk header includes the size of the header. The default value is false.

A *Chunk* object supports the following methods:

getname()

Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

getsize()

Returns the size of the chunk.

close()

Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise *OSError* if called after the *close()* method has been called. Before Python 3.3, they used to raise *IOError*, now an alias of *OSError*.

isatty()

Returns `False`.

seek(*pos*, *whence=0*)

Set the chunk's current position. The *whence* argument is optional and defaults to `0` (absolute file positioning); other values are `1` (seek relative to the current position) and `2` (seek relative to

[1] "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

the file's end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

tell()
Return the current position into the chunk.

read(*size=-1*)
Read at most *size* bytes from the chunk (less if the read hits the end of the chunk before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until the end of the chunk. An empty bytes object is returned when the end of the chunk is encountered immediately.

skip()
Skip to the end of the chunk. All further calls to *read()* for the chunk will return `b''`. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

23.6 colorsys — Conversions between color systems

Source code: Lib/colorsys.py

The *colorsys* module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

See also:

More information about color spaces can be found at http://poynton.ca/ColorFAQ.html and https://www.cambridgeincolour.com/tutorials/color-spaces.htm.

The *colorsys* module defines the following functions:

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
Convert the color from RGB coordinates to YIQ coordinates.

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
Convert the color from YIQ coordinates to RGB coordinates.

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
Convert the color from RGB coordinates to HLS coordinates.

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
Convert the color from HLS coordinates to RGB coordinates.

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
Convert the color from RGB coordinates to HSV coordinates.

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
Convert the color from HSV coordinates to RGB coordinates.

Example:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

23.7 imghdr — Determine the type of an image

Source code: Lib/imghdr.py

The *imghdr* module determines the type of image contained in a file or byte stream.

The *imghdr* module defines the following function:

imghdr.what(*filename*, *h=None*)
: Tests the image data contained in the file named by *filename*, and returns a string describing the image type. If optional *h* is provided, the *filename* is ignored and *h* is assumed to contain the byte stream to test.

  Changed in version 3.6: Accepts a *path-like object*.

The following image types are recognized, as listed below with the return value from *what()*:

Value	Image format
'rgb'	SGI ImgLib Files
'gif'	GIF 87a and 89a Files
'pbm'	Portable Bitmap Files
'pgm'	Portable Graymap Files
'ppm'	Portable Pixmap Files
'tiff'	TIFF Files
'rast'	Sun Raster Files
'xbm'	X Bitmap Files
'jpeg'	JPEG data in JFIF or Exif formats
'bmp'	BMP files
'png'	Portable Network Graphics
'webp'	WebP files
'exr'	OpenEXR Files

New in version 3.5: The *exr* and *webp* formats were added.

You can extend the list of file types *imghdr* can recognize by appending to this variable:

imghdr.tests
: A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When *what()* is called with a byte-stream, the file-like object will be None.

  The test function should return a string describing the image type if the test succeeded, or None if it failed.

Example:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

23.8 sndhdr — Determine type of sound file

Source code: Lib/sndhdr.py

The *sndhdr* provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a *namedtuple()*, containing five attributes: (`filetype`, `framerate`, `nchannels`, `nframes`, `sampwidth`). The value for *type* indicates the data type and will be one of the strings `'aifc'`, `'aiff'`, `'au'`, `'hcom'`, `'sndr'`, `'sndt'`, `'voc'`, `'wav'`, `'8svx'`, `'sb'`, `'ub'`, or `'ul'`. The *sampling_rate* will be either the actual value or `0` if unknown or difficult to decode. Similarly, *channels* will be either the number of channels or `0` if it cannot be determined or if the value is difficult to decode. The value for *frames* will be either the number of frames or `-1`. The last item in the tuple, *bits_per_sample*, will either be the sample size in bits or `'A'` for A-LAW or `'U'` for u-LAW.

`sndhdr.what`(*filename*)
: Determines the type of sound data stored in the file *filename* using *whathdr()*. If it succeeds, returns a namedtuple as described above, otherwise `None` is returned.

  Changed in version 3.5: Result changed from a tuple to a namedtuple.

`sndhdr.whathdr`(*filename*)
: Determines the type of sound data stored in a file based on the file header. The name of the file is given by *filename*. This function returns a namedtuple as described above on success, or `None`.

  Changed in version 3.5: Result changed from a tuple to a namedtuple.

23.9 `ossaudiodev` — Access to OSS-compatible audio devices

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

Changed in version 3.3: Operations in this module now raise *OSError* where *IOError* was raised.

See also:

Open Sound System Programmer's Guide the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see `<sys/soundcard.h>` on either Linux or FreeBSD for a listing.

ossaudiodev defines the following variables and functions:

`exception ossaudiodev.OSSAudioError`
: This exception is raised on certain errors. The argument is a string describing what went wrong.

  (If *ossaudiodev* receives an error from a system call such as `open()`, `write()`, or `ioctl()`, it raises *OSError*. Errors detected directly by *ossaudiodev* result in *OSSAudioError*.)

  (For backwards compatibility, the exception class is also available as `ossaudiodev.error`.)

`ossaudiodev.open`(*mode*)
`ossaudiodev.open`(*device*, *mode*)
: Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as `read()`, `write()`, and `fileno()` (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

  device is the audio device filename to use. If it is not specified, this module first looks in the environment variable `AUDIODEV` for a device to use. If not found, it falls back to `/dev/dsp`.

  mode is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is

a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older `linuxaudiodev` module which *ossaudiodev* supersedes.

`ossaudiodev.openmixer([device])`
Open a mixer device and return an OSS mixer device object. *device* is the mixer device filename to use. If it is not specified, this module first looks in the environment variable `MIXERDEV` for a device to use. If not found, it falls back to `/dev/mixer`.

23.9.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by *open()* define the following methods and (read-only) attributes:

`oss_audio_device.close()`
Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

`oss_audio_device.fileno()`
Return the file descriptor associated with the device.

`oss_audio_device.read(size)`
Read *size* bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block *read()* until the entire requested amount of data is available.

`oss_audio_device.write(data)`
Write a *bytes-like object* *data* to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire data is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written—see *writeall()*.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

`oss_audio_device.writeall(data)`
Write a *bytes-like object* *data* to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as *write()*; *writeall()* is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

Changed in version 3.5: Writable *bytes-like object* is now accepted.

Changed in version 3.2: Audio device objects also support the context management protocol, i.e. they can be used in a `with` statement.

The following methods each map to exactly one `ioctl()` system call. The correspondence is obvious: for example, `setfmt()` corresponds to the `SNDCTL_DSP_SETFMT` ioctl, and `sync()` to `SNDCTL_DSP_SYNC` (this can

be useful when consulting the OSS documentation). If the underlying `ioctl()` fails, they all raise *OSError*.

`oss_audio_device.nonblock()`
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

`oss_audio_device.getfmts()`
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

Format	Description
`AFMT_MU_LAW`	a logarithmic encoding (used by Sun `.au` files and `/dev/audio`)
`AFMT_A_LAW`	a logarithmic encoding
`AFMT_IMA_ADPCM`	a 4:1 compressed format defined by the Interactive Multimedia Association
`AFMT_U8`	Unsigned, 8-bit audio
`AFMT_S16_LE`	Signed, 16-bit audio, little-endian byte order (as used by Intel processors)
`AFMT_S16_BE`	Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
`AFMT_S8`	Signed, 8 bit audio
`AFMT_U16_LE`	Unsigned, 16-bit little-endian audio
`AFMT_U16_BE`	Unsigned, 16-bit big-endian audio

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support `AFMT_U8`; the most common format used today is `AFMT_S16_LE`.

`oss_audio_device.setfmt(`*format*`)`
Try to set the current audio format to *format*—see *getfmts()* for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an "audio format" of `AFMT_QUERY`.

`oss_audio_device.channels(`*nchannels*`)`
Set the number of output channels to *nchannels*. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

`oss_audio_device.speed(`*samplerate*`)`
Try to set the audio sampling rate to *samplerate* samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are:

Rate	Description
8000	default rate for `/dev/audio`
11025	speech recording
22050	
44100	CD quality audio (at 16 bits/sample and 2 channels)
96000	DVD quality audio (at 24 bits/sample)

`oss_audio_device.sync()`
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using *sync()*.

`oss_audio_device.reset()`
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling *reset()*.

oss_audio_device.post()
: Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.setparameters(*format*, *nchannels*, *samplerate*[, *strict=False*])
: Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format*, *nchannels*, and *samplerate* should be as specified in the *setfmt()*, *channels()*, and *speed()* methods. If *strict* is true, *setparameters()* checks to see if each parameter was actually set to the requested value, and raises *OSSAudioError* if not. Returns a tuple (*format*, *nchannels*, *samplerate*) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of *setfmt()*, *channels()*, and *speed()*).

  For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

  is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.bufsize()
: Returns the size of the hardware buffer, in samples.

oss_audio_device.obufcount()
: Returns the number of samples that are in the hardware buffer yet to be played.

oss_audio_device.obuffree()
: Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

oss_audio_device.closed
: Boolean indicating whether the device has been closed.

oss_audio_device.name
: String containing the name of the device file.

oss_audio_device.mode
: The I/O mode for the file, either `"r"`, `"rw"`, or `"w"`.

23.9.2 Mixer Device Objects

The mixer object provides two file-like methods:

oss_mixer_device.close()
: This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an *OSError*.

oss_mixer_device.fileno()
: Returns the file handle number of the open mixer device file.

Changed in version 3.2: Mixer objects also support the context management protocol.

The remaining methods are specific to audio mixing:

oss_mixer_device.controls()
: This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a subset of all available mixer controls—the SOUND_MIXER_* constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

oss_mixer_device.stereocontrols()
: Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with *controls()* to determine which).

See the code example for the *controls()* function for an example of getting data from a bitmask.

oss_mixer_device.reccontrols()
: Returns a bitmask specifying the mixer controls that may be used to record. See the code example for *controls()* for an example of reading from a bitmask.

oss_mixer_device.get(*control*)
: Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises *OSSAudioError* if an invalid control is specified, or *OSError* if an unsupported control is specified.

oss_mixer_device.set(*control*, *(left*, *right)*)
: Sets the volume for a given mixer control to (left,right). left and right must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises *OSSAudioError* if an invalid mixer control was specified, or if the specified volumes were out-of-range.

oss_mixer_device.get_recsrc()
: This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.set_recsrc(*bitmask*)
: Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises *OSError* if an invalid source was specified. To set the current recording source to the microphone input:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```