Python Audio Tools Documentation

Release 2.16

Brian Langenberger

October 23, 2010

CONTENTS

Contents:

CHAPTER

ONE

AUDIOTOOLS — THE BASE PYTHON AUDIO TOOLS MODULE

The audiotools module contains a number of useful base classes and functions upon which all of the other modules depend.

VERSION
The current Python Audio Tools version as a plain string.

AVAILABLE_TYPES
A tuple of AudioFile-compatible objects of available audio types. Note these are types available to audiotools, not necessarily available to the user - depending on whether the required binaries are installed or not.

Class	Format
AACAudio	AAC in ADTS container
AiffAudio	Audio Interchange File Format
ALACAudio	Apple Lossless
AuAudio	Sun Au
FlacAudio	Native Free Lossless Audio Codec
M4AAudio	AAC in M4A container
MP3Audio	MPEG-1 Layer 3
MP2Audio	MPEG-1 Layer 2
OggFlacAudio	Ogg Free Lossless Audio Codec
ShortenAudio	Shorten
SpeexAudio	Ogg Speex
VorbisAudio	Ogg Vorbis
WaveAudio	Waveform Audio File Format
WavPackAudio	WavPack

TYPE_MAP
A dictionary of type_name strings -> AudioFile values containing only types which have all required binaries installed.

BIN
A dictionary-like class for performing lookups of system binaries. This checks the system and user's config files and ensures that any redirected binaries are called from their proper location. For example, if the user has configured flac(1) to be run from /opt/flac/bin/flac

```
>>> BIN["flac"]
"/opt/flac/bin/flac"
```

This class also has a can_execute() method which returns True if the given binary is executable.

```
>>> BIN.can_execute(BIN["flac"])
True
```

open (*filename*)
Opens the given filename string and returns an AudioFile-compatible object. Raises UnsupportedFile if the file cannot identified or is not supported. Raises IOError if the file cannot be opened at all.

open_files (*filenames, [sorted, [messenger]]*)
Given a list of filename strings, returns a list of AudioFile-compatible objects which can be successfully opened. By default, they are returned sorted by album number and track number. If sorted is False, they are returned in the same order as they appear in the filenames list. If messenger is given, use that Messenger object to for warnings if files cannot be opened. Otherwise, such warnings are sent to stdout.

open_directory (*directory, [sorted, [messenger]]*)
Given a root directory, returns an iterator of all the AudioFile-compatible objects found via a recursive search of that directory. sorted, and messenger work as in open_files().

group_tracks (*audiofiles*)
Given an iterable collection of AudioFile-compatible objects, returns an iterator of objects grouped into lists by album. That is, all objects with the same album_name and album_number metadata fields will be returned in the same list on each pass.

filename_to_type (*path*)
Given a path, try to guess its AudioFile class based on its filename suffix. Raises UnknownAudioType if the suffix is unrecognized. Raises AmbiguousAudioType if more than one type of audio shares the same suffix.

transfer_data (*from_function, to_function*)
This function takes two functions, presumably analogous to write() and read() functions, respectively. It calls to_function on the object returned by calling from_function with an integer argument (presumably a string) until that object's length is 0.

```
>>> infile = open("input.txt","r")
>>> outfile = open("output.txt","w")
>>> transfer_data(infile.read,outfile.write)
>>> infile.close()
>>> outfile.close()
```

transfer_framelist_data (*pcmreader, to_function, [signed, [big_endian]]*)
A natural progression of transfer_data(), this function takes a PCMReader object and transfers the pcm.FrameList objects returned by its PCMReader.read() method to to_function after converting them to plain strings.

```
>>> pcm_data = audiotools.open("file.wav").to_pcm()
>>> outfile = open("output.pcm","wb")
>>> transfer_framelist_data(pcm_data,outfile)
>>> pcm_data.close()
>>> outfile.close()
```

pcm_cmp (*pcmreader1, pcmreader2*)
This function takes two PCMReader objects and compares their PCM output. Returns True if that output matches exactly, False if not.

stripped_pcm_cmp (*pcmreader1, pcmreader2*)
This function takes two PCMReader objects and compares their PCM output after stripping any 0 samples from the beginning and end of each. Returns True if the remaining output matches exactly, False if not.

pcm_frame_cmp (*pcmreader1, pcmreader2*)
This function takes two PCMReader objects and compares their PCM frame output. It returns the frame number of the first mismatch as an integer which begins at frame number 0. If the two streams match completely, it returns None.

pcm_split (*pcmreader, pcm_lengths*)
Takes a PCMReader object and list of PCM sample length integers. Returns an iterator of new PCMReader objects, each limited to the given lengths. The original pcmreader is closed upon the iterator's completion.

applicable_replay_gain (*audiofiles*)
Takes a list of AudioFile-compatible objects. Returns True if ReplayGain can be applied to those files based on their sample rate, number of channels, and so forth. Returns False if not.

calculate_replay_gain (*audiofiles*)
Takes a list of AudioFile-compatible objects. Returns an iterator of (audiofile, track_gain, track_peak, album_gain, album_peak) tuples or raises ValueError if a problem occurs during calculation.

read_metadata_file (*path*)
Given a path to a FreeDB XMCD file or MusicBrainz XML file, returns an AlbumMetaDataFile-compatible object or raises a MetaDataFileException if the file cannot be read or parsed correctly.

read_sheet (*filename*)
Reads a Cuesheet-compatible file such as toc.TOCFile or cue.Cuesheet or raises SheetException if the file cannot be opened, identified or parsed correctly.

find_glade_file (*glade_filename*)
Given a Glade filename, search various system directories for the full path to an existing file. Raises IOError if the file cannot be found.

1.1 AudioFile Objects

class **AudioFile** ()
The AudioFile class represents an audio file on disk, such as a FLAC file, MP3 file, WAVE file and so forth. It is not meant to be instantiated directly. Instead, functions such as open() will return AudioFile-compatible objects implementing the following methods.

class **is_type** (*file*)
Takes a file-like object with read() and seek() methods that's reset to the beginning of the stream. Returns True if the file is determined to be of the same type as this particular AudioFile implementation. Returns False if not.

bits_per_sample ()
Returns the number of bits-per-sample in this audio file as a positive integer.

channels ()
Returns the number of channels in this audio file as a positive integer.

channel_mask ()
Returns a ChannelMask object representing the channel assignment of this audio file. If the channel assignment is unknown or undefined, that ChannelMask object may have an undefined value.

sample_rate ()
Returns the sample rate of this audio file, in Hz, as a positive integer.

total_frames ()
Returns the total number of PCM frames in this audio file, as a non-negative integer.

cd_frames()
Returns the total number of CD frames in this audio file, as a non-negative integer. Each CD frame is 1/75th of a second.

lossless()
Returns `True` if the data in the audio file has been stored losslessly. Returns `False` if not.

set_metadata(*metadata*)
Takes a `MetaData`-compatible object and sets this audio file's metadata to that value, if possible. Raises `IOError` if a problem occurs when writing the file.

get_metadata()
Returns a `MetaData`-compatible object representing this audio file's metadata, or `None` if this file contains no metadata. Raises `IOError` if a problem occurs when reading the file.

delete_metadata()
Deletes the audio file's metadata, removing or unsetting tags as necessary. Raises `IOError` if a problem occurs when writing the file.

to_pcm()
Returns this audio file's PCM data as a `PCMReader`-compatible object. May return a `PCMReaderError` if an error occurs initializing the decoder.

class **from_pcm**(*filename, pcmreader, [compression=None]*)
Takes a filename string, `PCMReader`-compatible object and optional compression level string. Creates a new audio file as the same format as this audio class and returns a new `AudioFile`-compatible object. Raises `EncodingError` if a problem occurs during encoding.

In this example, we'll transcode `track.flac` to `track.mp3` at the default compression level:

```
>>> audiotools.MP3Audio.from_pcm("track.mp3",
...                              audiotools.open("track.flac").to_pcm())
```

to_wave(*wave_filename*)
Takes a filename string and creates a new RIFF WAVE file at that location. Raises `EncodingError` if a problem occurs during encoding.

class **from_wave**(*filename, wave_filename, [compression=None]*)
Takes a filename string of our new file, a wave_filename string of an existing RIFF WAVE file and an optional compression level string. Creates a new audio file as the same format as this audio class and returns a new `AudioFile`-compatible object. Raises `EncodingError` if a problem occurs during encoding.

class **supports_foreign_riff_chunks**()
Returns `True` if this `AudioFile` implementation supports storing non audio RIFF WAVE chunks. Returns `False` if not.

has_foreign_riff_chunks()
Returns `True` if this audio file contains non audio RIFF WAVE chunks. Returns `False` if not.

track_number()
Returns this audio file's track number as a non-negative integer. This method first checks the file's metadata values. If unable to find one, it then tries to determine a track number from the track's filename. If that method is also unsuccessful, it returns 0.

album_number()
Returns this audio file's album number as a non-negative integer. This method first checks the file's metadata values. If unable to find one, it then tries to determine an album number from the track's filename. If that method is also unsuccessful, it returns 0.

class **track_name** (*file_path, [track_metadata, [format]]*)
Given a file path string and optional MetaData-compatible object and Python format string, returns a filename string with the format string fields filled-in. If not provided by metadata, track_number and album_number will be determined from file_path, if possible. Raises UnsupportedTracknameField if the format string contains unsupported fields.

Currently supported fields are:

Field	Value
%(album_name)s	track_metadata.album_name
%(album_number)s	track_metadata.album_number
%(album_total)s	track_metadata.album_total
%(album_track_number)s	album_number combined with track_number
%(artist_name)s	track_metadata.artist_name
%(catalog)s	track_metadata.catalog
%(comment)s	track_metadata.comment
%(composer_name)s	track_metadata.composer_name
%(conductor_name)s	track_metadata.conductor_name
%(copyright)s	track_metadata.copyright
%(date)s	track_metadata.date
%(ISRC)s	track_metadata.ISRC
%(media)s	track_metadata.year
%(performer_name)s	track_metadata.performer_name
%(publisher)s	track_metadata.publisher
%(suffix)s	the AudioFile suffix
%(track_name)s	track_metadata.track_name
%(track_number)2.2d	track_metadata.track_number
%(track_total)s	track_metadata.track_total
%(year)s	track_metadata.year
%(basename)s	file_path basename without suffix

class **add_replay_gain** (*filenames*)
Given a list of filename strings of the same class as this AudioFile class, calculates and adds ReplayGain metadata to those files. Raises ValueError if some problem occurs during ReplayGain calculation or application.

class **can_add_replay_gain** ()
Returns True if this audio class supports ReplayGain and we have the necessary binaries to apply it. Returns False if not.

class **lossless_replay_gain** ()
Returns True if this audio class applies ReplayGain via a lossless process - such as by adding a metadata tag of some sort. Returns False if applying metadata modifies the audio file data itself.

replay_gain ()
Returns this audio file's ReplayGain values as a ReplayGain object, or None if this audio file has no values.

set_cuesheet (*cuesheet*)
Takes a cuesheet-compatible object with catalog(), IRSCs(), indexes() and pcm_lengths() methods and sets this audio file's embedded cuesheet to those values, if possible. Raises IOError if this AudioFile supports embedded cuesheets but some error occurred when writing the file.

get_cuesheet ()
Returns a cuesheet-compatible object with catalog(), IRSCs(), indexes() and pcm_lengths() methods or None if no cuesheet is embedded. Raises IOError if some error occurs when reading the file.

class **has_binaries** ()
Returns True if all the binaries necessary to implement this AudioFile-compatible class are present and

executable. Returns `False` if not.

1.2 MetaData Objects

class MetaData(*[track_name, [track_number, [track_total, [album_name, [artist_name, [performer_name, [composer_name, [conductor_name, [media, [ISRC, [catalog, [copyright, [publisher, [year, [data, [album_number, [album_total, [comment, [images]]]]]]]]]]]]]]]]]]]*)

The `MetaData` class represents an `AudioFile`'s non-technical metadata. It can be instantiated directly for use by the `set_metadata()` method. However, the `get_metadata()` method will typically return `MetaData`-compatible objects corresponding to the audio file's low-level metadata implementation rather than actual `MetaData` objects. Modifying fields within a `MetaData`-compatible object will modify its underlying representation and those changes will take effect should `set_metadata()` be called with that updated object.

The `images` argument, if given, should be an iterable collection of `Image`-compatible objects.

track_name
This individual track's name as a Unicode string.

track_number
This track's number within the album as an integer.

track_total
The total number of tracks on the album as an integer.

album_name
The name of this track's album as a Unicode string.

artist_name
The name of this track's original creator/composer as a Unicode string.

performer_name
The name of this track's performing artist as a Unicode string.

composer_name
The name of this track's composer as a Unicode string.

conductor_name
The name of this track's conductor as a Unicode string.

media
The album's media type, such as u"CD", u"tape", u"LP", etc. as a Unicode string.

ISRC
This track's ISRC value as a Unicode string.

catalog
This track's album catalog number as a Unicode string.

year
This track's album release year as a Unicode string.

date
This track's album recording date as a Unicode string.

album_number
This track's album number if it is one of a series of albums, as an integer.

album_total
The total number of albums within the set, as an integer.

comment
This track's comment as a Unicode string.

class **converted**(*metadata*)
Takes a MetaData-compatible object (or None) and returns a new MetaData object of the same class, or None. For instance, VorbisComment.converted() returns VorbisComment objects. The purpose of this classmethod is to offload metadata conversion to the metadata classes themselves. Therefore, by using the VorbisComment.converted() classmethod, the VorbisAudio class only needs to know how to handle VorbisComment metadata.

Why not simply handle all metadata using this high-level representation and avoid conversion altogether? The reason is that MetaData is often only a subset of what the low-level implementation can support. For example, a VorbisComment may contain the 'FOO' tag which has no analogue in MetaData's list of fields. But when passed through the VorbisComment.converted() classmethod, that 'FOO' tag will be preserved as one would expect.

The key is that performing:

```
>>> track.set_metadata(track.get_metadata())
```

should always round-trip properly and not lose any metadata values.

class **supports_images**()
Returns True if this MetaData implementation supports images. Returns False if not.

images()
Returns a list of Image-compatible objects this metadata contains.

front_covers()
Returns a subset of images() which are marked as front covers.

back_covers()
Returns a subset of images() which are marked as back covers.

leaflet_pages()
Returns a subset of images() which are marked as leaflet pages.

media_images()
Returns a subset of images() which are marked as media.

other_images()
Returns a subset of images() which are marked as other.

add_image(*image*)
Takes a Image-compatible object and adds it to this metadata's list of images.

delete_image(*image*)
Takes an Image from this class, as returned by images(), and removes it from this metadata's list of images.

merge(*new_metadata*)
Updates this metadata by replacing empty fields with those from new_metadata. Non-empty fields are left as-is.

1.3 AlbumMetaData Objects

class AlbumMetaData(*metadata_iter*)
This is a dictionary-like object of track_number -> MetaData values. It is designed to represent metadata returned by CD lookup services such as FreeDB or MusicBrainz.

metadata()
Returns a single MetaData object containing all the fields that are consistent across this object's collection of MetaData.

1.4 AlbumMetaDataFile Objects

class **AlbumMetaDataFile**(*album_name, artist_name, year, catalog, extra, track_metadata*)
This is an abstract parent class to audiotools.XMCD and audiotools.MusicBrainzReleaseXML. It represents a collection of album metadata as generated by the FreeDB or MusicBrainz services. Modifying fields within an AlbumMetaDataFile-compatible object will modify its underlying representation and those changes will be present when to_string() is called on the updated object. Note that audiotools.XMCD doesn't support the *catalog* field while audiotools.MusicBrainzReleaseXML doesn't support the *extra* fields.

album_name
The album's name as a Unicode string.

artist_name
The album's artist's name as a Unicode string.

year
The album's release year as a Unicode string.

catalog
The album's catalog number as a Unicode string.

extra
The album's extra information as a Unicode string.

__len__()
The total number of tracks on the album.

to_string()
Returns the on-disk representation of the file as a binary string.

class **from_string**(*string*)
Given a binary string, returns an AlbumMetaDataFile object of the same class. Raises MetaDataFileException if a parsing error occurs.

get_track(*index*)
Given a track index (starting from 0), returns a (*track_name*, *track_artist*, *track_extra*) tuple of Unicode strings. Raises IndexError if the requested track is out-of-bounds.

set_track(*index, track_name, track_artist, track_extra*)
Given a track index (starting from 0) and a set of Unicode strings, sets the appropriate track information. Raises IndexError if the requested track is out-of-bounds.

class **from_tracks**(*tracks*)
Given a set of AudioFile objects, returns an AlbumMetaDataFile object of the same class. All files are presumed to be from the same album.

class **from_cuesheet**(*cuesheet, total_frames, sample_rate, [metadata]*)
Given a Cuesheet-compatible object with catalog(), IRSCs(), indexes() and pcm_lengths() methods; *total_frames* and *sample_rate* integers; and an optional MetaData object of the entire album's metadata, returns an AlbumMetaDataFile object of the same class constructed from that data.

track_metadata(*track_number*)
Given a *track_number* (starting from 1), returns a MetaData object of that track's metadata.

Raises `IndexError` if the track is out-of-bounds.

get(*track_number, default*)
Given a *track_number* (starting from 1), returns a `MetaData` object of that track's metadata, or returns *default* if that track is not present.

track_metadatas()
Returns an iterator over all the `MetaData` objects in this file.

metadata()
Returns a single `MetaData` object of all consistent fields in this file. For example, if *album_name* is the same in all MetaData objects, the returned object will have that *album_name* value. If *track_name* differs, the returned object have a blank *track_name* field.

1.5 Image Objects

class Image(*data, mime_type, width, height, color_depth, color_count, description, type*)
This class is a container for image data.

data
A plain string of raw image bytes.

mime_type
A Unicode string of this image's MIME type, such as u'image/jpeg'

width
This image's width in pixels as an integer.

height
This image's height in pixels as an integer

color_depth
This image's color depth in bits as an integer. 24 for JPEG, 8 for GIF, etc.

color_count
For palette-based images, this is the number of colors the image contains as an integer. For non-palette images, this value is 0.

description
A Unicode string of this image's description.

type
An integer representing this image's type.

Value	Type
0	front cover
1	back cover
2	leaflet page
3	media
4	other

suffix()
Returns this image's typical filename suffix as a plain string. For example, JPEGs return `"jpg"`

type_string()
Returns this image's type as a plain string. For example, an image of type 0 returns `"Front Cover"`

class **new**(*image_data, description, type*)
Given a string of raw image bytes, a Unicode description string and image type integer, returns an `Image`-compatible object. Raises `InvalidImage` If unable to determine the image type from the data string.

thumbnail (*width, height, format*)
Given width and height integers and a format string (such as `"JPEG"`) returns a new `Image` object resized to those dimensions while retaining its original aspect ratio.

1.6 ReplayGain Objects

class ReplayGain (*track_gain, track_peak, album_gain, album_peak*)
This is a simple container for ReplayGain values.

track_gain
A float of a track's ReplayGain value.

track_peak
A float of a track's peak value, from 0.0 to 1.0

album_gain
A float of an album's ReplayGain value.

album_peak
A float of an album's peak value, from 0.0 to 1.0

1.7 PCMReader Objects

class PCMReader (*file, sample_rate, channels, channel_mask, bits_per_sample, [process, [signed, [big_endian]]]*)
This class wraps around file-like objects and generates `pcm.FrameList` objects on each call to `read()`. `sample_rate`, `channels`, `channel_mask` and `bits_per_sample` should be integers. `process` is a subprocess helper object which generates PCM data. `signed` is `True` if the generated PCM data is signed. `big_endian` is `True` if the generated PCM data is big-endian.

Note that `PCMReader`-compatible objects need only implement the `sample_rate`, `channels`, `channel_mask` and `bits_per_sample` fields. The rest are helpers for converting raw strings into `pcm.FrameList` objects.

sample_rate
The sample rate of this audio stream, in Hz, as a positive integer.

channels
The number of channels in this audio stream as a positive integer.

channel_mask
The channel mask of this audio stream as a non-negative integer.

bits_per_sample
The number of bits-per-sample in this audio stream as a positive integer.

read (*bytes*)
Try to read a `pcm.FrameList` object of size `bytes`, if possible. This method is *not* guaranteed to read that amount of bytes. It may return less, particularly at the end of an audio stream. It may even return FrameLists larger than requested. However, it must always return a non-empty FrameList until the end of the PCM stream is reached. May raise `IOError` if there is a problem reading the source file, or `ValueError` if the source file has some sort of error.

close ()
Closes the audio stream. If any subprocesses were used for audio decoding, they will also be closed and waited for their process to finish. May raise a `DecodingError`, typically indicating that a helper subprocess used for decoding has exited with an error.

1.7.1 PCMReaderError Objects

class PCMReaderError(*error_message, sample_rate, channels, channel_mask, bits_per_sample*)

This is a subclass of PCMReader which always returns empty pcm.FrameList objects and always raises a DecodingError with the given error_message when closed. The purpose of this is to postpone error generation so that all encoding errors, even those caused by unsuccessful decoding, are restricted to the from_pcm() classmethod which can then propagate the DecodingError error message to the user.

1.7.2 PCMConverter Objects

class PCMConverter(*pcmreader, sample_rate, channels, channel_mask, bits_per_sample*)

This class takes an existing PCMReader-compatible object along with a new set of sample_rate, channels, channel_mask and bits_per_sample values. Data from pcmreader is then automatically converted to the same format as those values.

sample_rate

If the new sample rate differs from pcmreader's sample rate, audio data is automatically resampled on each call to read().

channels

If the new number of channels is smaller than pcmreader's channel count, existing channels are removed or downmixed as necessary. If the new number of channels is larger, data from the first channel is duplicated as necessary to fill the rest.

channel_mask

If the new channel mask differs from pcmreader's channel mask, channels are removed as necessary such that the proper channel only outputs to the proper speaker.

bits_per_sample

If the new bits-per-sample differs from pcmreader's number of bits-per-sample, samples are shrunk or enlarged as necessary to cover the full amount of bits.

read()

This method functions the same as the PCMReader.read() method.

close()

This method functions the same as the PCMReader.close() method.

1.7.3 BufferedPCMReader Objects

class BufferedPCMReader(*pcmreader*)

This class wraps around an existing PCMReader object. Its calls to read() are guaranteed to return pcm.FrameList objects as close to the requested amount of bytes as possible without going over by buffering data internally.

The reason such behavior is not required is that we often don't care about the size of the individual FrameLists being passed from one routine to another. But on occasions when we need pcm.FrameList objects to be of a particular size, this class can accomplish that.

1.7.4 ReorderedPCMReader Objects

class ReorderedPCMReader(*pcmreader, channel_order*)

This class wraps around an existing PCMReader object. It takes a list of channel number integers (which should be the same as pcmreader's channel count) and reorders channels upon each call to read().

For example, to swap channels 0 and 1 in a stereo stream, one could do the following:

```
>>> reordered = ReorderedPCMReader(original, [1, 0])
```

Calls to `reordered.read()` will then have the left channel on the right side and vice versa.

1.7.5 PCMCat Objects

class `PCMCat` (*pcmreaders*)

This class wraps around an iterable group of `PCMReader` objects and concatenates their output into a single output stream.

Warning: `PCMCat` does not check that its input `PCMReader` objects all have the same sample rate, channels, channel mask or bits-per-sample. Mixing incompatible readers is likely to trigger undesirable behavior from any sort of processing - which often assumes data will be in a consistent format.

1.7.6 ReplayGainReader Objects

class `ReplayGainReader` (*pcmreader, gain, peak*)

This class wraps around an existing `PCMReader` object. It takes floating point `gain` and `peak` values and modifies the pcmreader's output as necessary to match those values. This has the effect of raising or lowering a stream's sound volume to ReplayGain's reference value.

1.8 ChannelMask Objects

class `ChannelMask` (*mask*)

This is an integer-like class that abstracts channel assignments into a set of bit fields.

Mask	Speaker
0x1	`front_left`
0x2	`front_right`
0x4	`front_center`
0x8	`low_frequency`
0x10	`back_left`
0x20	`back_right`
0x40	`front_left_of_center`
0x80	`front_right_of_center`
0x100	`back_center`
0x200	`side_left`
0x400	`side_right`
0x800	`top_center`
0x1000	`top_front_left`
0x2000	`top_front_center`
0x4000	`top_front_right`
0x8000	`top_back_left`
0x10000	`top_back_center`
0x20000	`top_back_right`

All channels in a `pcm.FrameList` will be in RIFF WAVE order as a sensible convention. But which channel corresponds to which speaker is decided by this mask. For example, a 4 channel PCMReader with the channel mask `0x33` corresponds to the bits `00110011`

Reading those bits from right to left (least significant first) the `front_left`, `front_right`, `back_left`, `back_right` speakers are set. Therefore, the PCMReader's 4 channel FrameLists are laid out as follows:

0. `front_left`

1. `front_right`

2. `back_left`

3. `back_right`

Since the `front_center` and `low_frequency` bits are not set, those channels are skipped in the returned FrameLists.

Many formats store their channels internally in a different order. Their `PCMReader` objects will be expected to reorder channels and set a `ChannelMask` matching this convention. And, their `from_pcm()` classmethods will be expected to reverse the process.

A `ChannelMask` of 0 is "undefined", which means that channels aren't assigned to *any* speaker. This is an ugly last resort for handling formats where multi-channel assignments aren't properly defined. In this case, a `from_pcm()` classmethod is free to assign the undefined channels any way it likes, and is under no obligation to keep them undefined when passing back out to `to_pcm()`

defined()
Returns `True` if this mask is defined.

undefined()
Returns `True` if this mask is undefined.

channels()
Returns the speakers this mask contains as a list of strings in the order they appear in the PCM stream.

index(*channel_name*)
Given a channel name string, returns the index of that channel within the PCM stream. For example:

```
>>> mask = ChannelMask(0xB)      #fL, fR, LFE, but no fC
>>> mask.index("low_frequency")
2
```

class **from_fields**(***fields*)
Takes channel names as function arguments and returns a `ChannelMask` object.

```
>>> mask = ChannelMask.from_fields(front_right=True,
...                                front_left=True,
...                                front_center=True)
>>> int(mask)
7
```

class **from_channels**(*channel_count*)
Takes a channel count integer and returns a `ChannelMask` object.

Warning: `from_channels()` *only* works for 1 and 2 channel counts and is meant purely as a convenience method for mono or stereo streams. All other values will trigger a `ValueError`

1.9 CDDA Objects

class CDDA(*device, [speed]*)
This class is used to access a CD-ROM device. It functions as a list of `CDTrackReader` objects, each repre-

senting a CD track and starting from index 1.

```
>>> cd = CDDA("/dev/cdrom")
>>> len(cd)
17
>>> cd[1]
<audiotools.CDTrackReader instance at 0x170def0>
>>> cd[17]
<audiotools.CDTrackReader instance at 0x1341b00>
```

length()
The length of the entire CD, in sectors.

first_sector()
The position of the first sector on the CD, typically 0.

last_sector()
The position of the last sector on the CD.

1.9.1 CDTrackReader Objects

class CDTrackReader(*cdda, track_number*)
These objects are usually retrieved from CDDA objects rather than instantiated directly. Each is a PCMReader-compatible object with a few additional methods specific to CD reading.

rip_log
A CDTrackLog object indicating cdparanoia's results from reading this track from the CD. This attribute should be checked only after the track has been fully read.

offset()
Returns the offset of this track within the CD, in sectors.

length()
Returns the total length of this track, in sectors.

1.9.2 CDTrackLog Objects

class CDTrackLog()
This is a dictionary-like object which should be retrieved from CDTrackReader rather than instantiated directly. Its `__str__()` method will return a human-readable collection of error statistics comparable to what's returned by the cdda2wav program.

1.10 ExecQueue Objects

class ExecQueue()
This is a class for executing multiple Python functions in parallel across multiple CPUs.

execute(*function, args, [kwargs]*)
Queues a Python function, list of arguments and optional dictionary of keyword arguments.

run(*[max_processes]*)
Executes all queued Python functions, running `max_processes` number of functions at a time until the entire queue is empty. This operates by forking a new subprocess per function, executing that function and then, regardless of the function's result, the child job performs an unconditional exit.

This means that any side effects of executed functions have no effect on ExecQueue's caller besides those which modify files on disk (encoding an audio file, for example).

1.11 Messenger Objects

class Messenger (*executable_name, options*)
This is a helper class for displaying program data, analogous to a primitive logging facility. It takes a raw `executable_name` string and `optparse.OptionParser` object. Its behavior changes depending on whether the `options` object's `verbosity` attribute is `"normal"`, `"debug"` or `"silent"`.

output (*string*)
Outputs Unicode `string` to stdout and adds a newline, unless `verbosity` level is `"silent"`.

partial_output (*string*)
Output Unicode `string` to stdout and flushes output so it is displayed, but does not add a newline. Does nothing if `verbosity` level is `"silent"`.

info (*string*)
Outputs Unicode `string` to stdout and adds a newline, unless `verbosity` level is `"silent"`.

partial_info (*string*)
Output Unicode `string` to stdout and flushes output so it is displayed, but does not add a newline. Does nothing if `verbosity` level is `"silent"`.

Note: What's the difference between `Messenger.output()` and `Messenger.info()`? `Messenger.output()` is for a program's primary data. `Messenger.info()` is for incidental information. For example, trackinfo uses `Messenger.output()` for what it displays since that output is its primary function. But track2track uses `Messenger.info()` for its lines of progress since its primary function is converting audio and tty output is purely incidental.

warning (*string*)
Outputs warning text, Unicode `string` and a newline to stderr, unless `verbosity` level is `"silent"`.

```
>>> m = audiotools.Messenger("audiotools",options)
>>> m.warning(u"Watch Out!")
*** Warning: Watch Out!
```

error (*string*)
Outputs error text, Unicode `string` and a newline to stderr.

```
>>> m.error(u"Fatal Error!")
*** Error: Fatal Error!
```

usage (*string*)
Outputs usage text, Unicode `string` and a newline to stderr.

```
>>> m.usage(u"<arg1> <arg2> <arg3>")
*** Usage: audiotools <arg1> <arg2> <arg3>
```

filename (*string*)
Takes a raw filename string and converts it to a Unicode string.

new_row()
This method begins the process of creating aligned table data output. It sets up a new row in our output table to which we can add columns of text which will be aligned automatically upon completion.

output_column(*string, [right_aligned]*)
This method adds a new Unicode string to the currently open row. If `right_aligned` is `True`, its text will be right-aligned when it is displayed. When you've finished with one row and wish to start on another, call `Messenger.new_row()` again.

blank_row()
This method adds a completely blank row to its table data. Note that the first row within an output table cannot be blank.

divider_row(*dividers*)
This method takes a list of vertical divider Unicode characters, one per output column, and multiplies those characters by their column width when displayed.

```
>>> m.new_row()
>>> m.output_column(u"foo")
>>> m.output_column(u" ")
>>> m.output_column(u"bar")
>>> m.divider_row([u"-",u" ",u"-"])
>>> m.new_row()
>>> m.output_column(u"test")
>>> m.output_column(u" ")
>>> m.output_column(u"column")
>>> m.output_rows()
foo  bar
---- ------
test column
```

output_rows()
Formats and displays the entire table data through the `Messenger.output()` method (which will do nothing if `verbosity` level is `"silent"`).

```
>>> m.new_row()
>>> m.output_column(u"a",True)
>>> m.output_column(u" : ",True)
>>> m.output_column(u"This is some test data")
>>> m.new_row()
>>> m.output_column(u"ab",True)
>>> m.output_column(u" : ",True)
>>> m.output_column(u"Another row of test data")
>>> m.new_row()
>>> m.output_column(u"abc",True)
>>> m.output_column(u" : ",True)
>>> m.output_column(u"The final row of test data")
>>> m.output_rows()
  a : This is some test data
 ab : Another row of test data
abc : The final row of test data
```

ansi(*string, codes*)
Takes a Unicode string and list of ANSI SGR code integers. If `stdout` is to a TTY, returns a Unicode string formatted with those codes. If not, the string is returned as is. Codes can be taken from the many predefined values in the `Messenger` class. Note that not all output terminals are guaranteed to support all ANSI escape codes.

ansi_err(*string, codes*)
This is identical to `Messenger.ansi`, but it checks whether `stderr` is a TTY instead of `stdout`.

Code	Effect
Messenger.RESET	resets current codes
Messenger.BOLD	bold font
Messenger.FAINT	faint font
Messenger.ITALIC	italic font
Messenger.UNDERLINE	underline text
Messenger.BLINK_SLOW	blink slowly
Messenger.BLINK_FAST	blink quickly
Messenger.REVERSE	reverse text
Messenger.STRIKEOUT	strikeout text
Messenger.FG_BLACK	foreground black
Messenger.FG_RED	foreground red
Messenger.FG_GREEN	foreground green
Messenger.FG_YELLOW	foreground yellow
Messenger.FG_BLUE	foreground blue
Messenger.FG_MAGENTA	foreground magenta
Messenger.FG_CYAN	foreground cyan
Messenger.FG_WHITE	foreground write
Messenger.BG_BLACK	background black
Messenger.BG_RED	background red
Messenger.BG_GREEN	background green
Messenger.BG_YELLOW	background yellow
Messenger.BG_BLUE	background blue
Messenger.BG_MAGENTA	background magenta
Messenger.BG_CYAN	background cyan
Messenger.BG_WHITE	background white

CHAPTER

TWO

AUDIOTOOLS.PCM — THE PCM FRAMELIST MODULE

The audiotools.pcm module contains the FrameList and FloatFrameList classes for handling blobs of raw data. These classes are immutable and list-like, but provide several additional methods and attributes to aid in processing PCM data.

from_list (*list, channels, bits_per_sample, is_signed*)
Given a list of integer values, a number of channels, the amount of bits-per-sample and whether the samples are signed, returns a new FrameList object with those values. Raises ValueError if a FrameList cannot be built from those values.

```
>>> f = from_list([-1,0,1,2],2,16,True)
>>> list(f)
[-1, 0, 1, 2]
```

from_frames (*frame_list*)
Given a list of FrameList objects, returns a new FrameList whose values are built from those objects. Raises ValueError if any of the objects are longer than 1 PCM frame, their number of channels are not consistent or their bits_per_sample are not consistent.

```
>>> l = [from_list([-1,0],2,16,True),
...      from_list([ 1,2],2,16,True)]
>>> f = from_frames(l)
>>> list(f)
[-1, 0, 1, 2]
```

from_channels (*frame_list*)
Given a list of FrameList objects, returns a new FrameList whose values are built from those objects. Raises ValueError if any of the objects are wider than 1 channel, their number of frames are not consistent or their bits_per_sample are not consistent.

```
>>> l = [from_list([-1,1],1,16,True),
...      from_list([ 0,2],1,16,True)]
>>> f = from_channels(l)
>>> list(f)
[-1, 0, 1, 2]
```

from_float_frames (*float_frame_list*)
Given a list of FloatFrameList objects, returns a new FloatFrameList whose values are built from those objects. Raises ValueError if any of the objects are longer than 1 PCM frame or their number of channels are not consistent.

```
>>> l = [FloatFrameList([-1.0,0.0],2),
...      FloatFrameList([ 0.5,1.0],2)]
>>> f = from_float_frames(l)
>>> list(f)
[-1.0, 0.0, 0.5, 1.0]
```

from_float_channels (*float_frame_list*)
Given a list of `FloatFrameList` objects, returns a new `FloatFrameList` whose values are built from those objects. Raises `ValueError` if any of the objects are wider than 1 channel or their number of frames are not consistent.

```
>>> l = [FloatFrameList([-1.0,0.5],1),
...      FloatFrameList([ 0.0,1.0],1)]
>>> f = from_float_channels(l)
>>> list(f)
[-1.0, 0.0, 0.5, 1.0]
```

2.1 FrameList Objects

class FrameList (*string, channels, bits_per_sample, is_big_endian, is_signed*)
This class implements a PCM FrameList, which can be envisioned as a 2D array of signed integers where each row represents a PCM frame of samples and each column represents a channel.

During initialization, `string` is a collection of raw bytes, `bits_per_sample` is an integer and `is_big_endian` and `is_signed` are booleans. This provides a convenient way to transforming raw data from file-like objects into `FrameList` objects. Once instantiated, a `FrameList` object is immutable.

frames
The amount of PCM frames within this object, as a non-negative integer.

channels
The amount of channels within this object, as a positive integer.

bits_per_sample
The size of each sample in bits, as a positive integer.

frame (*frame_number*)
Given a non-negative `frame_number` integer, returns the samples at the given frame as a new `FrameList` object. This new FrameList will be a single frame long, but have the same number of channels and bits_per_sample as the original. Raises `IndexError` if one tries to get a frame number outside this FrameList's boundaries.

channel (*channel_number*)
Given a non-negative `channel_number` integer, returns the samples at the given channel as a new `FrameList` object. This new FrameList will be a single channel wide, but have the same number of frames and bits_per_sample as the original. Raises `IndexError` if one tries to get a channel number outside this FrameList's boundaries.

split (*frame_count*)
Returns a pair of `FrameList` objects. The first contains up to `frame_count` number of PCM frames. The second contains the remainder. If `frame_count` is larger than the number of frames in the FrameList, the first will contain all of the frames and the second will be empty.

to_float ()
Converts this object's values to a new `FloatFrameList` object by transforming all samples to the range -1.0 to 1.0.

to_bytes (*is_big_endian, is_signed*)
Given `is_big_endian` and `is_signed` booleans, returns a plain string of raw PCM data. This is much like the inverse of `FrameList`'s initialization routine.

frame_count (*bytes*)
A convenience method which converts a given byte count to the maximum number of frames those bytes could contain, or a minimum of 1.

```
>>> FrameList("",2,16,False,True).frame_count(8)
2
```

2.2 FloatFrameList Objects

class FloatFrameList (*floats, channels*)
This class implements a FrameList of floating point samples, which can be envisioned as a 2D array of signed floats where each row represents a PCM frame of samples, each column represents a channel and each value is within the range of -1.0 to 1.0.

During initialization, `floats` is a list of float values and `channels` is an integer number of channels.

frames
The amount of PCM frames within this object, as a non-negative integer.

channels
The amount of channels within this object, as a positive integer.

frame (*frame_number*)
Given a non-negative `frame_number` integer, returns the samples at the given frame as a new `FloatFrameList` object. This new FloatFrameList will be a single frame long, but have the same number of channels as the original. Raises `IndexError` if one tries to get a frame number outside this FloatFrameList's boundaries.

channel (*channel_number*)
Given a non-negative `channel_number` integer, returns the samples at the given channel as a new `FloatFrameList` object. This new FloatFrameList will be a single channel wide, but have the same number of frames as the original. Raises `IndexError` if one tries to get a channel number outside this FloatFrameList's boundaries.

split (*frame_count*)
Returns a pair of `FloatFrameList` objects. The first contains up to `frame_count` number of PCM frames. The second contains the remainder. If `frame_count` is larger than the number of frames in the FloatFrameList, the first will contain all of the frames and the second will be empty.

to_int (*bits_per_sample*)
Given a `bits_per_sample` integer, converts this object's floating point values to a new `FrameList` object.

CHAPTER

THREE

AUDIOTOOLS.RESAMPLE — THE RESAMPLER MODULE

The audiotools.resample module contains a resampler for modifying the sample rate of PCM data. This class is not usually instantiated directly; instead, one can use audiotools.PCMConverter which calculates the resampling ratio and handles unprocessed samples automatically.

3.1 Resampler Objects

class Resampler (*channels, ratio, quality*)

This class performs the actual resampling and maintains the resampler's state. channels is the number of channels in the stream being resampled. ratio is the new sample rate divided by the current sample rate. quality is an integer value between 0 and 4, where 0 is the best quality.

For example, to convert a 2 channel, 88200Hz audio stream to 44100Hz, one starts by building a resampler as follows:

```
>>> resampler = Resampler(2, float(44100) / float(88200), 0)
```

process (*float_frame_list, last*)

Given a FloatFrameList object and whether this is the last chunk of PCM data from the stream, returns a pair of new FloatFrameList objects. The first is the processed samples at the new rate. The second is a set of unprocessed samples which must be pushed through again on the next call to process().

CHAPTER

FOUR

AUDIOTOOLS.REPLAYGAIN — THE REPLAYGAIN CALCULATION MODULE

The audiotools.replaygain module contains the ReplayGain class for calculating the ReplayGain gain and peak values for a set of PCM data.

4.1 ReplayGain Objects

class ReplayGain (*sample_rate*)
This class performs ReplayGain calculation for a stream of the given sample_rate. Raises ValueError if the sample rate is not supported.

update (*frame_list*)
Takes a pcm.FrameList object and updates our ongoing ReplayGain calculation. Raises ValueError if some error occurs during calculation.

title_gain()
Returns a pair of floats. The first is the calculated gain value since our last call to title_gain(). The second is the calculated peak value since our last call to title_gain().

album_gain()
Returns a pair of floats. The first is the calculated gain value of the entire stream. The first is the calculated peak value of the entire stream.

CHAPTER

FIVE

AUDIOTOOLS.CDIO — THE CD INPUT/OUTPUT MODULE

The audiotools.cdio module contains the CDDA class for accessing raw CDDA data. One does not typically use this module directly. Instead, the audiotools.CDDA class provides encapsulation to hide many of these low-level details.

5.1 CDDA Objects

class CDDA(*device*)

This class is used to access a specific CD-ROM device, which should be given as a string such as "/dev/cdrom" during instantiation.

Note that audio CDs are accessed by sectors, each 1/75th of a second long - or 588 PCM frames. Thus, many of this object's methods take and return sector integer values.

total_tracks()

Returns the total number of tracks on the CD as an integer.

```
>>> cd = CDDA("/dev/cdrom")
>>> cd.total_tracks()
17
```

track_offsets(*track_number*)

Given a track_number integer (starting from 1), returns a pair of sector values. The first is the track's first sector on the CD. The second is the track's last sector on the CD.

```
>>> cd.track_offsets(1)
(0, 15774)
>>> cd.track_offsets(2)
(15775, 31836)
```

first_sector()

Returns the first sector of the entire CD as an integer, typically 0.

```
>>> cd.first_sector()
0
```

last_sector()

Returns the last sector of the entire CD as an integer.

```
>>> cd.last_sector()
240449
```

length_in_seconds()
Returns the length of the entire CD in seconds as an integer.

```
>>> cd.length_in_seconds()
3206
```

track_type(*track_number*)
Given a track_number integer (starting from 1), returns the type of track it is as an integer.

set_speed(*speed*)
Sets the CD-ROM's reading speed to the new integer value.

seek(*sector*)
Sets our current position on the CD to the given sector. For example, to begin reading audio data from the second track:

```
>>> cd.track_offsets(2)[0]
15775
>>> cd.seek(15775)
```

read_sector()
Reads a single sector from the CD as a `pcm.FrameList` object and moves our current read position ahead by 1.

```
>>> f = cd.read_sector()
>>> f
<pcm.FrameList object at 0x2ca16f0>
>>> len(f)
1176
```

read_sectors(*sectors*)
Given a number of sectors, reads as many as possible from the CD as a `pcm.FrameList` object and moves our current read position ahead by that many sectors.

```
>>> f = cd.read_sectors(10)
>>> f
<pcm.FrameList object at 0x7f022e0d6c60>
>>> len(f)
11760
```

set_read_callback(*function*)
Sets a global callback function which takes two integer values as arguments. The second argument is a cdparanoia value corresponding to errors fixed, if any:

Value	CDParanoia Value	Meaning
0	PARANOIA_CB_READ	Read off adjust ???
1	PARANOIA_CB_VERIFY	Verifying jitter
2	PARANOIA_CB_FIXUP_EDGE	Fixed edge jitter
3	PARANOIA_CB_FIXUP_ATOM	Fixed atom jitter
4	PARANOIA_CB_SCRATCH	Unsupported
5	PARANOIA_CB_REPAIR	Unsupported
6	PARANOIA_CB_SKIP	Skip exhausted retry
7	PARANOIA_CB_DRIFT	Skip exhausted retry
8	PARANOIA_CB_BACKOFF	Unsupported
9	PARANOIA_CB_OVERLAP	Dynamic overlap adjust
10	PARANOIA_CB_FIXUP_DROPPED	Fixed dropped bytes
11	PARANOIA_CB_FIXUP_DUPED	Fixed duplicate bytes
12	PARANOIA_CB_READERR	Hard read error

CHAPTER

SIX

AUDIOTOOLS.CUE — THE CUESHEET PARSING MODULE

The audiotools.cue module contains the Cuesheet class used for parsing and building cuesheet files representing CD images.

read_cuesheet(*filename*)
Takes a filename string and returns a new Cuesheet object. Raises CueException if some error occurs when reading the file.

exception CueException
A subclass of audiotools.SheetException raised when some parsing or reading error occurs when reading a cuesheet file.

6.1 Cuesheet Objects

class Cuesheet()
This class is used to represent a .cue file. It is not meant to be instantiated directly but returned from the read_cuesheet() function. The __str__() value of a Cuesheet corresponds to a formatted file on disk.

catalog()
Returns the cuesheet's catalog number as a plain string, or None if the cuesheet contains no catalog number.

single_file_type()
Returns True if the cuesheet is formatted for a single input file. Returns False if the cuesheet is formatted for several individual tracks.

indexes()
Returns an iterator of index lists. Each index is a tuple of CD sectors corresponding to a track's offset on disk.

pcm_lengths(*total_length*)
Takes the total length of the entire CD in PCM frames. Returns a list of PCM frame lengths for all audio tracks within the cuesheet. This list of lengths can be used to split a single CD image file into several individual tracks.

ISRCs()
Returns a dictionary of track_number -> ISRC values for all tracks whose ISRC value is not empty.

class **file**(*sheet, filename*)
Takes a Cuesheet-compatible object with catalog(), indexes(), ISRCs() methods along with a filename string. Returns a new Cuesheet object. This is used to convert other sort of Cuesheet-like objects into actual Cuesheets.

CHAPTER

SEVEN

AUDIOTOOLS.TOC — THE TOC FILE PARSING MODULE

The audiotools.toc module contains the TOCFile class used for parsing and building TOC files representing CD images.

read_tocfile(*filename*)
Takes a filename string and returns a new TOCFile object. Raises TOCException if some error occurs when reading the file.

exception TOCException
A subclass of audiotools.SheetException raised when some parsing or reading error occurs when reading a TOC file.

7.1 TOCFile Objects

class TOCFile()
This class is used to represent a .toc file. It is not meant to be instantiated directly but returned from the read_tocfile() function.

catalog()
Returns the TOC file's catalog number as a plain string, or None if the TOC file contains no catalog number.

indexes()
Returns an iterator of index lists. Each index is a tuple of CD sectors corresponding to a track's offset on disk.

pcm_lengths(*total_length*)
Takes the total length of the entire CD in PCM frames. Returns a list of PCM frame lengths for all audio tracks within the TOC file. This list of lengths can be used to split a single CD image file into several individual tracks.

ISRCs()
Returns a dictionary of track_number -> ISRC values for all tracks whose ISRC value is not empty.

class **file**(*sheet, filename*)
Takes a cue.Cuesheet-compatible object with catalog(), indexes(), ISRCs() methods along with a filename string. Returns a new TOCFile object. This is used to convert other sort of Cuesheet-like objects into actual TOC files.

CHAPTER

EIGHT

META DATA FORMATS

Although it's more convenient to manipulate the high-level `audiotools.MetaData` base class, one sometimes needs to be able to view and modify the low-level implementation also.

8.1 ApeTag

class ApeTag(*tags, [tag_length]*)

This is an APEv2 tag used by the WavPack, Monkey's Audio and Musepack formats, among others. During initialization, it takes a list of `ApeTagItem` objects and an optional length integer (typically set only by `get_metadata()` methods which already know the tag's total length). It can then be manipulated like a regular Python dict with keys as strings and values as `ApeTagItem` objects. Note that this is also a `audiotools.MetaData` subclass with all of the same methods.

For example:

```
>>> tag = ApeTag([ApeTagItem(0,False,'Title',u'Track Title'.encode('utf-8'))])
>>> tag.track_name
u'Track Title'
>>> tag['Title']
ApeTagItem(0,False,'Title','Track Title')
>>> tag['Title'] = ApeTagItem(0,False,'Title',u'New Title'.encode('utf-8'))
>>> tag.track_name
u'New Title'
>>> tag.track_name = u'Yet Another Title'
>>> tag['Title']
ApeTagItem(0,False,'Title','Yet Another Title')
```

The fields are mapped between `ApeTag` and `audiotools.MetaData` as follows:

APEv2	Metadata
Title	track_name
Track	track_number/track_total
Media	album_number/album_total
Album	album_name
Artist	artist_name
Performer	performer_name
Composer	composer_name
Conductor	conductor_name
ISRC	ISRC
Catalog	catalog
Copyright	copyright
Publisher	publisher
Year	year
Record Date	date
Comment	comment

Note that Track and Media may be "/"-separated integer values where the first is the current number and the second is the total number.

```
>>> tag = ApeTag([ApeTagItem(0,False,'Track',u'1'.encode('utf-8'))])
>>> tag.track_number
1
>>> tag.track_total
0
>>> tag = ApeTag([ApeTagItem(0,False,'Track',u'2/3'.encode('utf-8'))])
>>> tag.track_number
2
>>> tag.track_total
3
```

class **read**(*file*)
Takes an open file object and returns an ApeTag object of that file's APEv2 data, or None if the tag cannot be found.

build()
Returns this tag's complete APEv2 data as a string.

class ApeTagItem(*item_type, read_only, key, data*)
This is the container for ApeTag data. item_type is an integer with one of the following values:

1	UTF-8 data
2	binary data
3	external data
4	reserved

read_only is a boolean set to True if the tag-item is read-only. key is an ASCII string. data is a regular Python string (not unicode).

build()
Returns this tag item's data as a string.

class **binary**(*key, data*)
A convenience classmethod which takes strings of key and value data and returns a populated ApeTagItem object of the appropriate type.

class **external**(*key, data*)
A convenience classmethod which takes strings of key and value data and returns a populated ApeTagItem object of the appropriate type.

class **string**(*key, unicode*)
A convenience classmethod which takes a key string and value unicode and returns a populated ApeTagItem object of the appropriate type.

8.2 FLAC

class FlacMetaData(*blocks*)
This is a FLAC tag which is prepended to FLAC and Ogg FLAC files. It is initialized with a list of FlacMetaDataBlock objects which it stores internally in one of several fields. It also supports all audiotools.MetaData methods.

For example:

```
>>> tag = FlacMetaData([FlacMetaDataBlock(
...                     type=4,
...                     data=FlacVorbisComment({u'TITLE':[u'Track Title']}).build())])
>>> tag.track_name
u'Track Title'
>>> tag.vorbis_comment[u'TITLE']
[u'Track Title']
>>> tag.vorbis_comment = a.FlacVorbisComment({u'TITLE':[u'New Track Title']})
>>> tag.track_name
u'New Track Title'
```

Its fields are as follows:

streaminfo
A FlacMetaDataBlock object containing raw STREAMINFO data. Since FLAC's set_metadata() method will override this attribute as necessary, one will rarely need to parse it or set it.

vorbis_comment
A FlacVorbisComment object containing text data such as track name and artist name. If the FLAC file doesn't have a VORBISCOMMENT block, FlacMetaData will set an empty one at initialization time which will then be written out by a call to set_metadata().

cuesheet
A FlacCueSheet object containing CUESHEET data, or None.

image_blocks
A list of FlacPictureComment objects, each representing a PICTURE block. The list may be empty.

extra_blocks
A list of raw FlacMetaDataBlock objects containing any unknown or unsupported FLAC metadata blocks. Note that padding is not stored here. PADDING blocks are discarded at initialization time and then re-created as needed by calls to set_metadata().

metadata_blocks()
Returns an iterator over all the current blocks as FlacMetaDataBlock-compatible objects and without any padding block at the end.

build(*[padding_size]*)
Returns a string of this FlacMetaData object's contents.

class FlacMetaDataBlock(*type, data*)
This is a simple container for FLAC metadata block data. type is one of the following block type integers:

0	STREAMINFO
1	PADDING
2	APPLICATION
3	SEEKTABLE
4	VORBIS_COMMENT
5	CUESHEET
6	PICTURE

`data` is a string.

build_block(*[last]*)
Returns the entire metadata block as a string, including the header. Set `last` to 1 to indicate this is the final metadata block in the stream.

class **FlacVorbisComment**(*vorbis_data, [vendor_string]*)
This is a subclass of `VorbisComment` modified to be FLAC-compatible. It utilizes the same initialization information and field mappings.

build_block(*[last]*)
Returns the entire metadata block as a string, including the header. Set `last` to 1 to indicate this is the final metadata block in the stream.

class **FlacPictureComment**(*type, mime_type, description, width, height, color_depth, color_count, data*)
This is a subclass of `audiotools.Image` with additional methods to make it FLAC-compatible.

build()
Returns this picture data as a block data string, without the metadata block headers. Raises `FlacMetaDataBlockTooLarge` if the size of its picture data exceeds 16777216 bytes.

build_block(*[last]*)
Returns the entire metadata block as a string, including the header. Set `last` to 1 to indicate this is the final metadata block in the stream.

class **FlacCueSheet**(*container, [sample_rate]*)
This is a `audiotools.cue.Cuesheet`-compatible object with `catalog()`, `ISRCs()`, `indexes()` and `pcm_lengths()` methods, in addition to those needed to make it FLAC metadata block compatible. Its `container` argument is an `audiotools.Con.Container` object which is returned by calling `FlacCueSheet.CUESHEET.parse()` on a raw input data string.

build_block(*[last]*)
Returns the entire metadata block as a string, including the header. Set `last` to 1 to indicate this is the final metadata block in the stream.

class **converted**(*sheet, total_frames, [sample_rate]*)
Takes another `audiotools.cue.Cuesheet`-compatible object and returns a new `FlacCueSheet` object.

8.3 ID3v1

class **ID3v1Comment**(*metadata*)
This is an ID3v1 tag which is often appended to MP3 files. During initialization, it takes a tuple of 6 values - in the same order as returned by `ID3v1Comment.read_id3v1_comment()`. It can then be manipulated like a regular Python list, in addition to the regular `audiotools.MetaData` methods. However, since ID3v1 is a nearly complete subset of `audiotools.MetaData` (the genre integer is the only field not represented), there's little need to reference its items by index directly.

For example:

```
>>> tag = ID3v1Comment((u'Track Title',u'',u'',u'',u'',1))
>>> tag.track_name
u'Track Title'
>>> tag[0] = u'New Track Name'
>>> tag.track_name
u'New Track Name'
```

Fields are mapped between `ID3v1Comment` and `audiotools.MetaData` as follows:

Index	Metadata
0	`track_name`
1	`artist_name`
2	`album_name`
3	`year`
4	`comment`
5	`track_number`

build_tag()
Returns this tag as a string.

class **build_id3v1**(*song_title, artist, album, year, comment, track_number*)
A convenience method which takes several unicode strings (except for `track_number`, an integer) and returns a complete ID3v1 tag as a string.

class **read_id3v1_comment**(*filename*)
Takes an MP3 filename string and returns a tuple of that file's ID3v1 tag data, or tag data with empty fields if no ID3v1 tag is found.

8.4 ID3v2.2

class ID3v22Comment(*frames*)
This is an ID3v2.2 tag, one of the three ID3v2 variants used by MP3 files. During initialization, it takes a list of `ID3v22Frame`-compatible objects. It can then be manipulated like a regular Python dict with keys as 3 character frame identifiers and values as lists of `ID3v22Frame` objects - since each frame identifier may occur multiple times.

For example:

```
>>> tag = ID3v22Comment([ID3v22TextFrame('TT2',0,u'Track Title')])
>>> tag.track_name
u'Track Title'
>>> tag['TT2']
[<audiotools.__id3__.ID3v22TextFrame instance at 0x1004c17a0>]
>>> tag['TT2'] = [ID3v22TextFrame('TT2',0,u'New Track Title')]
>>> tag.track_name
u'New Track Title'
```

Fields are mapped between ID3v2.2 frame identifiers, `audiotools.MetaData` and `ID3v22Frame` objects as follows:

Identifier	MetaData	Object
TT2	track_name	ID3v22TextFrame
TRK	track_number/track_total	ID3v22TextFrame
TPA	album_number/album_total	ID3v22TextFrame
TAL	album_name	ID3v22TextFrame
TP1	artist_name	ID3v22TextFrame
TP2	performer_name	ID3v22TextFrame
TP3	conductor_name	ID3v22TextFrame
TCM	composer_name	ID3v22TextFrame
TMT	media	ID3v22TextFrame
TRC	ISRC	ID3v22TextFrame
TCR	copyright	ID3v22TextFrame
TPB	publisher	ID3v22TextFrame
TYE	year	ID3v22TextFrame
TRD	date	ID3v22TextFrame
COM	comment	ID3v22ComFrame
PIC	images()	ID3v22PicFrame

class ID3v22Frame(*frame_id, data*)
This is the base class for the various ID3v2.2 frames. frame_id is a 3 character string and data is the frame's contents as a string.

build()
Returns the frame's contents as a string of binary data.

class **parse**(*container*)
Given a audiotools.Con.Container object with data parsed from audiotools.ID3v22Frame.FRAME, returns an ID3v22Frame or one of its subclasses, depending on the frame identifier.

class ID3v22TextFrame(*frame_id, encoding, string*)
This is a container for textual data. frame_id is a 3 character string, string is a unicode string and encoding is one of the following integers representing a text encoding:

0	Latin-1
1	UCS-2

__int__()
Returns the first integer portion of the frame data as an int.

total()
Returns the integer portion of the frame data after the first slash as an int. For example:

```
>>> tag['TRK'] = [ID3v22TextFrame('TRK',0,u'1/2')]
>>> tag['TRK']
[<audiotools.__id3__.ID3v22TextFrame instance at 0x1004c6830>]
>>> int(tag['TRK'][0])
1
>>> tag['TRK'][0].total()
2
```

class **from_unicode**(*frame_id, s*)
A convenience method for building ID3v22TextFrame objects from a frame identifier and unicode string. Note that if frame_id is "COM", this will build an ID3v22ComFrame object instead.

class ID3v22ComFrame(*encoding, language, short_description, content*)
This frame is for holding a potentially large block of comment data. encoding is the same as in text frames:

0	Latin-1
1	UCS-2

`language` is a 3 character string, such as `"eng"` for English. `short_description` and `content` are unicode strings.

class **from_unicode**(*s*)
A convenience method for building `ID3v22ComFrame` objects from a unicode string.

class ID3v22PicFrame(*data, format, description, pic_type*)
This is a subclass of `audiotools.Image`, in addition to being an ID3v2.2 frame. `data` is a string of binary image data. `format` is a 3 character unicode string identifying the image type:

`u"PNG"`	PNG
`u"JPG"`	JPEG
`u"BMP"`	Bitmap
`u"GIF"`	GIF
`u"TIF"`	TIFF

`description` is a unicode string. `pic_type` is an integer representing one of the following:

0	Other
1	32x32 pixels 'file icon' (PNG only)
2	Other file icon
3	Cover (front)
4	Cover (back)
5	Leaflet page
6	Media (e.g. label side of CD)
7	Lead artist / Lead performer / Soloist
8	Artist / Performer
9	Conductor
10	Band / Orchestra
11	Composer
12	Lyricist / Text writer
13	Recording Location
14	During recording
15	During performance
16	Movie / Video screen capture
17	A bright colored fish
18	Illustration
19	Band / Artist logotype
20	Publisher / Studio logotype

type_string()
Returns the `pic_type` as a plain string.

class **converted**(*image*)
Given an `audiotools.Image` object, returns a new `ID3v22PicFrame` object.

8.5 ID3v2.3

class ID3v23Comment(*frames*)
This is an ID3v2.3 tag, one of the three ID3v2 variants used by MP3 files. During initialization, it takes a list of `ID3v23Frame`-compatible objects. It can then be manipulated like a regular Python dict with keys as 4 character frame identifiers and values as lists of `ID3v23Frame` objects - since each frame identifier may occur multiple times.

For example:

```
>>> tag = ID3v23Comment([ID3v23TextFrame('TIT2',0,u'Track Title')])
>>> tag.track_name
u'Track Title'
>>> tag['TIT2']
[<audiotools.__id3__.ID3v23TextFrame instance at 0x1004c6680>]
>>> tag['TIT2'] = [ID3v23TextFrame('TIT2',0,u'New Track Title')]
>>> tag.track_name
u'New Track Title'
```

Fields are mapped between ID3v2.3 frame identifiers, `audiotools.MetaData` and `ID3v23Frame` objects as follows:

Identifier	MetaData	Object
TIT2	track_name	ID3v23TextFrame
TRCK	track_number/track_total	ID3v23TextFrame
TPOS	album_number/album_total	ID3v23TextFrame
TALB	album_name	ID3v23TextFrame
TPE1	artist_name	ID3v23TextFrame
TPE2	performer_name	ID3v23TextFrame
TPE3	conductor_name	ID3v23TextFrame
TCOM	composer_name	ID3v23TextFrame
TMED	media	ID3v23TextFrame
TSRC	ISRC	ID3v23TextFrame
TCOP	copyright	ID3v23TextFrame
TPUB	publisher	ID3v23TextFrame
TYER	year	ID3v23TextFrame
TRDA	date	ID3v23TextFrame
COMM	comment	ID3v23ComFrame
APIC	images()	ID3v23PicFrame

class ID3v23Frame (*frame_id, data*)
This is the base class for the various ID3v2.3 frames. `frame_id` is a 4 character string and `data` is the frame's contents as a string.

build()
Returns the frame's contents as a string of binary data.

class **parse** (*container*)
Given a `audiotools.Con.Container` object with data parsed from `audiotools.ID3v23Frame.FRAME`, returns an `ID3v23Frame` or one of its subclasses, depending on the frame identifier.

class ID3v23TextFrame (*frame_id, encoding, string*)
This is a container for textual data. `frame_id` is a 4 character string, `string` is a unicode string and `encoding` is one of the following integers representing a text encoding:

0	Latin-1
1	UCS-2

__int__()
Returns the first integer portion of the frame data as an int.

total()
Returns the integer portion of the frame data after the first slash as an int. For example:

```
>>> tag['TRAK'] = [ID3v23TextFrame('TRAK',0,u'3/4')]
>>> tag['TRAK']
```

```
[<audiotools.__id3__.ID3v23TextFrame instance at 0x1004c17a0>]
>>> int(tag['TRAK'][0])
3
>>> tag['TRAK'][0].total()
4
```

class **from_unicode**(*frame_id, s*)
A convenience method for building `ID3v23TextFrame` objects from a frame identifier and unicode string. Note that if `frame_id` is `"COMM"`, this will build an `ID3v23ComFrame` object instead.

class ID3v23ComFrame(*encoding, language, short_description, content*)
This frame is for holding a potentially large block of comment data. `encoding` is the same as in text frames:

0	Latin-1
1	UCS-2

`language` is a 3 character string, such as `"eng"` for english. `short_description` and `content` are unicode strings.

class **from_unicode**(*s*)
A convenience method for building `ID3v23ComFrame` objects from a unicode string.

class ID3v23PicFrame(*data, mime_type, description, pic_type*)
This is a subclass of `audiotools.Image`, in addition to being an ID3v2.3 frame. `data` is a string of binary image data. `mime_type` is a string of the image's MIME type, such as `"image/jpeg"`.

`description` is a unicode string. `pic_type` is an integer representing one of the following:

0	Other
1	32x32 pixels 'file icon' (PNG only)
2	Other file icon
3	Cover (front)
4	Cover (back)
5	Leaflet page
6	Media (e.g. label side of CD)
7	Lead artist / Lead performer / Soloist
8	Artist / Performer
9	Conductor
10	Band / Orchestra
11	Composer
12	Lyricist / Text writer
13	Recording Location
14	During recording
15	During performance
16	Movie / Video screen capture
17	A bright colored fish
18	Illustration
19	Band / Artist logotype
20	Publisher / Studio logotype

class **converted**(*image*)
Given an `audiotools.Image` object, returns a new `ID3v23PicFrame` object.

8.6 ID3v2.4

class ID3v24Comment (*frames*)

This is an ID3v2.4 tag, one of the three ID3v2 variants used by MP3 files. During initialization, it takes a list of `ID3v24Frame`-compatible objects. It can then be manipulated like a regular Python dict with keys as 4 character frame identifiers and values as lists of `ID3v24Frame` objects - since each frame identifier may occur multiple times.

For example:

```
>>> import audiotools as a
>>> tag = ID3v24Comment([ID3v24TextFrame('TIT2',0,u'Track Title')])
>>> tag.track_name
u'Track Title'
>>> tag['TIT2']
[<audiotools.__id3__.ID3v24TextFrame instance at 0x1004c17a0>]
>>> tag['TIT2'] = [ID3v24TextFrame('TIT2',0,'New Track Title')]
>>> tag.track_name
u'New Track Title'
```

Fields are mapped between ID3v2.4 frame identifiers, `audiotools.MetaData` and `ID3v24Frame` objects as follows:

Identifier	MetaData	Object
TIT2	track_name	ID3v24TextFrame
TRCK	track_number/track_total	ID3v24TextFrame
TPOS	album_number/album_total	ID3v24TextFrame
TALB	album_name	ID3v24TextFrame
TPE1	artist_name	ID3v24TextFrame
TPE2	performer_name	ID3v24TextFrame
TPE3	conductor_name	ID3v24TextFrame
TCOM	composer_name	ID3v24TextFrame
TMED	media	ID3v24TextFrame
TSRC	ISRC	ID3v24TextFrame
TCOP	copyright	ID3v24TextFrame
TPUB	publisher	ID3v24TextFrame
TYER	year	ID3v24TextFrame
TRDA	date	ID3v24TextFrame
COMM	comment	ID3v24ComFrame
APIC	images()	ID3v24PicFrame

class ID3v24Frame (*frame_id, data*)

This is the base class for the various ID3v2.3 frames. `frame_id` is a 4 character string and `data` is the frame's contents as a string.

build()

Returns the frame's contents as a string of binary data.

class **parse** (*container*)

Given a `audiotools.Con.Container` object with data parsed from `audiotools.ID3v24Frame.FRAME`, returns an `ID3v24Frame` or one of its subclasses, depending on the frame identifier.

class ID3v24TextFrame (*frame_id, encoding, string*)

This is a container for textual data. `frame_id` is a 4 character string, `string` is a unicode string and `encoding` is one of the following integers representing a text encoding:

0	Latin-1
1	UTF-16
2	UTF-16BE
3	UTF-8

__int__()
Returns the first integer portion of the frame data as an int.

total()
Returns the integer portion of the frame data after the first slash as an int. For example:

```
>>> tag['TRAK'] = [ID3v24TextFrame('TRAK',0,u'5/6')]
>>> tag['TRAK']
[<audiotools.__id3__.ID3v24TextFrame instance at 0x1004c17a0>]
>>> int(tag['TRAK'][0])
5
>>> tag['TRAK'][0].total()
6
```

class **from_unicode**(*frame_id, s*)
A convenience method for building `ID3v24TextFrame` objects from a frame identifier and unicode string. Note that if `frame_id` is `"COMM"`, this will build an `ID3v24ComFrame` object instead.

class ID3v24ComFrame(*encoding, language, short_description, content*)
This frame is for holding a potentially large block of comment data. `encoding` is the same as in text frames:

0	Latin-1
1	UTF-16
2	UTF-16BE
3	UTF-8

`language` is a 3 character string, such as `"eng"` for english. `short_description` and `content` are unicode strings.

class **from_unicode**(*s*)
A convenience method for building `ID3v24ComFrame` objects from a unicode string.

class ID3v24PicFrame(*data, mime_type, description, pic_type*)
This is a subclass of `audiotools.Image`, in addition to being an ID3v2.4 frame. `data` is a string of binary image data. `mime_type` is a string of the image's MIME type, such as `"image/jpeg"`.

`description` is a unicode string. `pic_type` is an integer representing one of the following:

0	Other
1	32x32 pixels 'file icon' (PNG only)
2	Other file icon
3	Cover (front)
4	Cover (back)
5	Leaflet page
6	Media (e.g. label side of CD)
7	Lead artist / Lead performer / Soloist
8	Artist / Performer
9	Conductor
10	Band / Orchestra
11	Composer
12	Lyricist / Text writer
13	Recording Location
14	During recording
15	During performance
16	Movie / Video screen capture
17	A bright colored fish
18	Illustration
19	Band / Artist logotype
20	Publisher / Studio logotype

class **converted**(*image*)
Given an `audiotools.Image` object, returns a new `ID3v24PicFrame` object.

8.7 ID3 Comment Pair

Often, MP3 files are tagged with both an ID3v2 comment and an ID3v1 comment for maximum compatibility. This class encapsulates both comments into a single class.

class ID3CommentPair(*id3v2_comment, id3v1_comment*)
`id3v2_comment` is an `ID3v22Comment`, `ID3v23Comment` or `ID3v24Comment`. `id3v1_comment` is an `ID3v1Comment`. When getting `audiotools.MetaData` attributes, the ID3v2 comment is used by default. Set attributes are propagated to both. For example:

```
>>> tag = ID3CommentPair(ID3v23Comment([ID3v23TextFrame('TIT2',0,u'Title 1')]),
...                      ID3v1Comment((u'Title 2',u'',u'',u'',u'',1)))
>>> tag.track_name
u'Title 1'
>>> tag.track_name = u'New Track Title'
>>> unicode(tag.id3v2['TIT2'][0])
u'New Track Title'
>>> tag.id3v1[0]
u'New Track Title'
```

id3v2
The embedded `ID3v22Comment`, `ID3v23Comment` or `ID3v24Comment`

id3v1
The embedded `ID3v1Comment`

8.8 M4A

class M4AMetaData (*ilst_atoms*)

This is the metadata format used by QuickTime-compatible formats such as M4A and Apple Lossless. Due to its relative complexity, M4AMetaData's implementation is more low-level than others. During initialization, it takes a list of ILST_Atom-compatible objects. It can then be manipulated like a regular Python dict with keys as 4 character atom name strings and values as a list of ILST_Atom objects. It is also a audiotools.MetaData subclass. Note that ilst atom objects are relatively opaque and easier to handle via convenience builders.

As an example:

```
>>> tag = M4AMetaData(M4AMetaData.text_atom(chr(0xA9) + 'nam',u'Track Name'))
>>> tag.track_name
u'Track Name'
>>> tag[chr(0xA9) + 'nam']
[ILST_Atom('\xa9nam',[__Qt_Atom__('data','\x00\x00\x00\x01\x00\x00\x00\x00Track Name',0)])]
>>> tag[chr(0xA9) + 'nam'] = M4AMetaData.text_atom(chr(0xA9) + 'nam',u'New Track Name')
>>> tag.track_name
u'New Track Name'
```

Fields are mapped between M4AMetaData, audiotools.MetaData and iTunes as follows:

M4AMetaData	MetaData	iTunes
"\xA9nam"	track_name	Name
"\xA9ART"	artist_name	Artist
"\xA9day"	year	Year
"trkn"	track_number/track_total	Track Number
"disk"	album_number/album_total	Album Number
"\xA9alb"	album_name	Album
"\xA9wrt"	composer_name	Composer
"\xA9cmt"	comment	Comment
"cprt"	copyright	

Note that several of the 4 character keys are prefixed by the non-ASCII byte 0xA9.

to_atom (*previous_meta*)

This takes the previous M4A meta atom as a string and returns a new __Qt_Atom__ object of our new meta atom with any non-ilst atoms ported from the old atom to the new atom.

class **binary_atom** (*key, value*)

Takes a 4 character atom name key and binary string value. Returns a 1 element ILST_Atom list suitable for adding to our internal dictionary.

class **text_atom** (*key, value*)

Takes a 4 character atom name key and unicode value. Returns a 1 element ILST_Atom list suitable for adding to our internal dictionary.

class **trkn_atom** (*track_number, track_total*)

Takes track number and track total integers (the trkn key is assumed). Returns a 1 element ILST_Atom list suitable for adding to our internal dictionary.

class **disk_atom** (*disk_number, disk_total*)

Takes album number and album total integers (the disk key is assumed). Returns a 1 element ILST_Atom list suitable for adding to our internal dictionary.

class **covr_atom** (*image_data*)

Takes a binary string of cover art data (the covr key is assumed). Returns a 1 element ILST_Atom list suitable for adding to our internal dictionary.

class ILST_Atom(*type, sub_atoms*)

This is initialized with a 4 character atom type string and a list of `__Qt_Atom__`-compatible sub-atom objects (typically a single `data` atom containing the metadata field's value). It's less error-prone to use `M4AMetaData`'s convenience classmethods rather than building `ILST_Atom` objects by hand.

Its `__unicode__()` method is particularly useful because it parses its sub-atoms and returns a human-readable value depending on whether it contains textual data or not.

8.9 Vorbis Comment

class VorbisComment(*vorbis_data, [vendor_string]*)

This is a VorbisComment tag used by FLAC, Ogg FLAC, Ogg Vorbis, Ogg Speex and other formats in the Ogg family. During initialization `vorbis_data` is a dictionary whose keys are unicode strings and whose values are lists of unicode strings - since each key in a Vorbis Comment may occur multiple times with different values. The optional `vendor_string` unicode string is typically handled by `get_metadata()` and `set_metadata()` methods, but it can also be accessed via the `vendor_string` attribute. Once initialized, `VorbisComment` can be manipulated like a regular Python dict in addition to its standard `audiotools.MetaData` methods.

For example:

```
>>> tag = VorbisComment({u'TITLE':[u'Track Title']})
>>> tag.track_name
u'Track Title'
>>> tag[u'TITLE']
[u'New Title']
>>> tag[u'TITLE'] = [u'New Title']
>>> tag.track_name
u'New Title'
```

Fields are mapped between `VorbisComment` and `audiotools.MetaData` as follows:

VorbisComment	Metadata
TITLE	track_name
TRACKNUMBER	track_number
TRACKTOTAL	track_total
DISCNUMBER	album_number
DISCTOTAL	album_total
ALBUM	album_name
ARTIST	artist_name
PERFORMER	performer_name
COMPOSER	composer_name
CONDUCTOR	conductor_name
SOURCE MEDIUM	media
ISRC	ISRC
CATALOG	catalog
COPYRIGHT	copyright
PUBLISHER	publisher
DATE	year
COMMENT	comment

Note that if the same key is used multiple times, the metadata attribute only indicates the first one:

```
>>> tag = VorbisComment({u'TITLE':[u'Title1',u'Title2']})
>>> tag.track_name
u'Title1'
```

build()
Returns this object's complete Vorbis Comment data as a string.

CHAPTER

NINE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

A

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W