Python Audio Tools Documentation

Release 2.17

Brian Langenberger

April 03, 2011

CONTENTS

1 audiotools — the Base Python Audio Tools Module **3**
- 1.1 AudioFile Objects . . . 5
- 1.2 MetaData Objects . . . 9
- 1.3 AlbumMetaData Objects . . . 11
- 1.4 AlbumMetaDataFile Objects . . . 11
- 1.5 Image Objects . . . 12
- 1.6 ReplayGain Objects . . . 13
- 1.7 PCMReader Objects . . . 13
- 1.8 ChannelMask Objects . . . 16
- 1.9 CDDA Objects . . . 17
- 1.10 DVDAudio Objects . . . 18
- 1.11 ExecQueue Objects . . . 20
- 1.12 ExecProgressQueue Objects . . . 21
- 1.13 Messenger Objects . . . 22
- 1.14 ProgressDisplay Objects . . . 25

2 audiotools.pcm — the PCM FrameList Module **27**
- 2.1 FrameList Objects . . . 28
- 2.2 FloatFrameList Objects . . . 29

3 audiotools.resample — the Resampler Module **31**
- 3.1 Resampler Objects . . . 31

4 audiotools.replaygain — the ReplayGain Calculation Module **33**
- 4.1 ReplayGain Objects . . . 33
- 4.2 ReplayGainReader Objects . . . 33

5 audiotools.cdio — the CD Input/Output Module **35**
- 5.1 CDDA Objects . . . 35

6 audiotools.cue — the Cuesheet Parsing Module **37**
- 6.1 Cuesheet Objects . . . 37

7 audiotools.toc — the TOC File Parsing Module **39**
- 7.1 TOCFile Objects . . . 39

8 audiotools.player — the Audio Player Module **41**
- 8.1 Player Objects . . . 41
- 8.2 CDPlayer Objects . . . 42
- 8.3 AudioOutput Objects . . . 42

9 Meta Data Formats **45**
- 9.1 ApeTag . . . 45
- 9.2 FLAC . . . 46

9.3 ID3v1 . . . 48
9.4 ID3v2.2 . . . 49
9.5 ID3v2.3 . . . 51
9.6 ID3v2.4 . . . 53
9.7 ID3 Comment Pair . . . 55
9.8 M4A . . . 55
9.9 Vorbis Comment . . . 56

10 Indices and tables **59**

Python Module Index **61**

Index **63**

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.

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

audiotools.**AVAILABLE_TYPES**
A tuple of AudioFile-compatible classes 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

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

audiotools.**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
```

audiotools.**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.

audiotools.**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.

audiotools.**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().

audiotools.**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.

audiotools.**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.

audiotools.**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()
```

audiotools.**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()
```

audiotools.**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.

audiotools.**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.

audiotools.**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. May raise IOError or ValueError if problems occur during reading.

audiotools.**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.

audiotools.**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.

audiotools.**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.

audiotools.**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.

audiotools.**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.

audiotools.**to_pcm_progress**(*audiofile*, *progress*)
Given an AudioFile-compatible object and progress function, returns a PCMReaderProgress object of that object's PCM stream.

If progress is None, the audiofile's PCM stream is returned as-is.

1.1 AudioFile Objects

class audiotools.**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 with the following attributes and methods.

AudioFile.**NAME**
The name of the format as a string. This is how the format is referenced by utilities via the *-t* option, and must be unique among all formats.

AudioFile.**SUFFIX**
The default file suffix as a string. This is used by the %(suffix)s format field in the track_name() classmethod, and by the filename_to_type() function for inferring the file format from its name. However, it need not be unique among all formats.

AudioFile.**COMPRESSION_MODES**
A tuple of valid compression level strings, for use with the from_pcm() and convert() methods. If the format has no compression levels, this tuple will be empty.

AudioFile.**DEFAULT_COMPRESSION**
A string of the default compression level to use with from_pcm() and convert(), if none is given. This is *not* the default compression indicated in the user's configuration file; it is a hard-coded value of last resort.

AudioFile.**COMPRESSION_DESCRIPTIONS**
A dict of compression descriptions, as unicode strings. The key is a valid compression mode string. Not all compression modes need have a description; some may be left blank.

AudioFile.**BINARIES**
A tuple of binary strings required by the format. For example, the Vorbis format may require "oggenc" and "oggdec" in order to be available for the user.

AudioFile.**REPLAYGAIN_BINARIES**
A tuple of binary strings required for ReplayGain application. For example, the Vorbis format may require "vorbisgain" in order to use the add_replay_gain() classmethod. This tuple may be empty if the format requires no binaries or has no ReplayGain support.

classmethod AudioFile.**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.

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

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

`AudioFile.`**`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.

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

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

`AudioFile.`**`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.

`AudioFile.`**`seconds_length`**`()`
Returns the length of this audio file as a `decimal.Decimal` number of seconds.

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

`AudioFile.`**`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.

`AudioFile.`**`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.

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

`AudioFile.`**`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.

classmethod `AudioFile.`**`from_pcm`**`(`*filename*, *pcmreader*[, *compression*]`)`
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())
```

`AudioFile.`**`convert`**`(`*filename*, *target_class*[, *compression*[, *progress*]]`)`
Takes a filename string, `AudioFile` subclass and optional compression level string. Creates a new audio file and returns an object of the same class. 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.open("track.flac").convert("track.mp3",
...                                       audiotools.MP3Audio)
```

Why have both a `convert` method as well as `to_pcm`/`from_pcm` methods? Although the former is often implemented using the latter, the pcm methods alone contain only raw audio data. By comparison, the

`convert` method has information about what is the file is being converted to and can transfer other side data if necessary.

For example, if .wav file with non-audio RIFF chunks is converted to WavPack, this method will preserve those chunks:

```
>>> audiotools.open("chunks.wav").convert("chunks.wv",
...                                       audiotools.WavPackAudio)
```

whereas the `to_pcm`/`from_pcm` method alone will not.

The optional `progress` argument is a function which takes two integer arguments: `amount_processed` and `total_amount`. If supplied, this function is called at regular intervals during the conversion process and may be used to indicate the current status to the user. Note that these numbers are only meaningful when compared to one another; `amount` may represent PCM frames, bytes or anything else. The only restriction is that `total_amount` will remain static during processing and `amount_processed` will progress from 0 to `total_amount`.

```
>>> def print_progress(x, y):
...   print "%d%%" % (x * 100 / y)
...
>>> audiotools.open("track.flac").convert("track.wv",
...                                       audiotools.WavPackAudio,
...                                       progress=print_progress)
```

`AudioFile.`**`verify`**`(`[*progress*]`)`

Verifies the track for correctness. Returns `True` if verification is successful. Raises an `InvalidFile` subclass if some problem is detected. If the file has built-in checksums or other error detection capabilities, this method checks those values to ensure it has not been damaged in some way.

The optional `progress` argument functions identically to the one provided to `convert()`. That is, it takes a two integer argument function which is called at regular intervals to indicate the status of verification.

`AudioFile.`**`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.

`AudioFile.`**`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.

classmethod `AudioFile.`**`track_name`**`(`*file_path*[, *track_metadata*[, *format*[, *suffix*]]]`)`

Given a file path string and optional `MetaData`-compatible object a UTF-8 encoded Python format string, and an ASCII-encoded suffix 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

classmethod AudioFile.**add_replay_gain**(*filenames*[, *progress*])
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. progress, if indicated, is a function which takes two arguments that is called as needed during ReplayGain application to indicate progress - identical to the argument used by convert().

classmethod AudioFile.**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.

classmethod AudioFile.**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.

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

AudioFile.**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.

AudioFile.**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.

classmethod AudioFile.**has_binaries**(*system_binaries*)
Takes the audiotools.BIN object of system binaries. Returns True if all the binaries necessary to implement this AudioFile-compatible class are present and executable. Returns False if not.

1.1.1 WaveContainer Objects

This is an abstract AudioFile subclass suitable for extending by formats that store RIFF WAVE chunks internally, such as Wave, FLAC, WavPack and Shorten. It overrides the AudioFile.convert() method such that any stored chunks are transferred properly from one file to the next. This is accomplished by implementing three additional methods.

class audiotools.**WaveContainer**

WaveContainer.**to_wave**(*wave_filename*[, *progress*])
Creates a Wave file with the given filename string from our data, with any stored chunks intact. progress, if given, functions identically to the AudioFile.convert() method. May raise EndodingError if some problem occurs during encoding.

classmethod WaveContainer.**from_wave**(*filename*, *wave_filename*[, *compression*[, *progress*]])
Like AudioFile.from_pcm(), creates a file with our class at the given filename string, from the given wave_filename string and returns a new object of our class. compression is an optional compression level string and progress functions identically to that of AudioFile.convert(). May raise EndodingError if some problem occurs during encoding.

WaveContainer.**has_foreign_riff_chunks**()
Returns True if our object has non-audio RIFF WAVE chunks.

1.1.2 AiffContainer Objects

Much like WaveContainer, this is an abstract AudioFile subclass suitable for extending by formats that store AIFF chunks internally, such as AIFF, FLAC and Shorten. It overrides the AudioFile.convert() method such that any stored chunks are transferred properly from one file to the next. This is accomplished by implementing three additional methods.

class audiotools.**AiffContainer**

AiffContainer.**to_aiff**(*aiff_filename*[, *progress*])
Creates an AIFF file with the given filename string from our data, with any stored chunks intact. progress, if given, functions identically to the AudioFile.convert() method. May raise EndodingError if some problem occurs during encoding.

classmethod AiffContainer.**from_aiff**(*filename*, *aiff_filename*[, *compression*[, *progress*]])
Like AudioFile.from_pcm(), creates a file with our class at the given filename string, from the given aiff_filename string and returns a new object of our class. compression is an optional compression level string and progress functions identically to that of AudioFile.convert(). May raise EndodingError if some problem occurs during encoding.

AiffContainer.**has_foreign_aiff_chunks**()
Returns True if our object has non-audio AIFF chunks.

1.2 MetaData Objects

class audiotools.**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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

MetaData.**comment**
This track's comment as a Unicode string.

classmethod MetaData.**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.

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

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

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

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

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

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

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

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

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

MetaData.**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 audiotools.**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.

AlbumMetaData.**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 audiotools.**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.

AlbumMetaDataFile.**album_name**
The album's name as a Unicode string.

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

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

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

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

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

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

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

AlbumMetaDataFile.**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.

AlbumMetaDataFile.**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.

classmethod AlbumMetaDataFile.**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.

classmethod AlbumMetaDataFile.**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.

AlbumMetaDataFile.**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.

AlbumMetaDataFile.**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.

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

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

Image.**data**
A plain string of raw image bytes.

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

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

Image.**height**
This image's height in pixels as an integer

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

`Image.`**`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.

`Image.`**`description`**
A Unicode string of this image's description.

`Image.`**`type`**
An integer representing this image's type.

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

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

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

classmethod `Image.`**`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.

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

`ReplayGain.`**`track_gain`**
A float of a track's ReplayGain value.

`ReplayGain.`**`track_peak`**
A float of a track's peak value, from 0.0 to 1.0

`ReplayGain.`**`album_gain`**
A float of an album's ReplayGain value.

`ReplayGain.`**`album_peak`**
A float of an album's peak value, from 0.0 to 1.0

1.7 PCMReader Objects

class `audiotools.`**`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.

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

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

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

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

PCMReader.**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.

PCMReader.**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 audiotools.**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 audiotools.**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.

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

PCMConverter.**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.

PCMConverter.**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.

PCMConverter.**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.

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

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

1.7.3 BufferedPCMReader Objects

class audiotools.**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 audiotools.**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 audiotools.**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 PCMReaderWindow Objects

class audiotools.**PCMReaderWindow**(*pcmreader*, *initial_offset*, *total_pcm_frames*)
This class wraps around an existing PCMReader object and truncates or extends its samples as needed. initial_offset, if positive, indicates how many PCM frames to truncate from the beginning of the stream. If negative, the beginning of the stream is padded by that many PCM frames - all of which have a value of 0. total_pcm_frames indicates the total length of the stream as a non-negative number of PCM frames. If shorter than the actual length of the PCM reader's stream, the reader is truncated. If longer, the stream is extended by as many PCM frames as needed. Again, padding frames have a value of 0.

1.7.7 LimitedPCMReader Objects

class audiotools.**LimitedPCMReader**(*buffered_pcmreader*, *total_pcm_frames*)
This class wraps around an existing BufferedPCMReader and ensures that no more than total_pcm_frames will be read from that stream by limiting reads to it.

Note: `PCMReaderWindow` is designed primarly for handling sample offset values in a `CDTrackReader`, or for skipping a potentially large number of samples in a stream. `LimitedPCMReader` is designed for splitting a stream into several smaller streams without losing any PCM frames.

Which to use for a given situation depends on whether one cares about consuming the samples outside of the sub-reader or not.

1.7.8 PCMReaderProgress Objects

class `audiotools.`**`PCMReaderProgress`**(*pcmreader*, *total_frames*, *progress*)

This class wraps around an existing `PCMReader` object and generates periodic updates to a given `progress` function. `total_frames` indicates the total number of PCM frames in the PCM stream.

```
>>> progress_display = SingleProgressDisplay(Messenger("audiotools"), u"encoding file")
>>> pcmreader = source_audiofile.to_pcm()
>>> source_frames = source_audiofile.total_frames()
>>> target_audiofile = AudioType.from_pcm("target_filename",
...                                       PCMReaderProgress(pcmreader,
...                                                         source_frames,
...                                                         progress_display.update))
```

1.8 ChannelMask Objects

class `audiotools.`**`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()

ChannelMask.**defined**()
Returns True if this mask is defined.

ChannelMask.**undefined**()
Returns True if this mask is undefined.

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

ChannelMask.**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
```

classmethod ChannelMask.**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
```

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

If True, perform_logging indicates that track reads should generate CDTrackLog entries. Otherwise, no logging is performed.

Warning: perform_logging also determines the level of multithreading allowed during CD reading. If logging is active, CDTrackReader's read method will block all other threads until the read is complete. If logging is inactive, a read will not block other threads. This is an unfortunate necessity due to libcdio's callback mechanism implementation.

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

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

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

1.9.1 CDTrackReader Objects

class audiotools.**CDTrackReader**(*cdda*, *track_number*[, *perform_logging*])
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.

CDTrackReader.**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.

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

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

1.9.2 CDTrackLog Objects

class audiotools.**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 DVDAudio Objects

class audiotools.**DVDAudio**(*audio_ts_path*[, *device*])
This class is used to access a DVD-Audio. It contains a collection of titlesets. Each titleset contains a list of DVDATitle objects, and each DVDATitle contains a list of DVDATrack objects. audio_ts_path is the path to the DVD-Audio's AUDIO_TS directory, such as /media/cdrom/AUDIO_TS. device is the path to the DVD-Audio's mount device, such as /dev/cdrom.

For example, to access the 3rd DVDATrack object of the 2nd DVDATitle of the first titleset, one can simply perform the following:

```
>>> track = DVDAudio(path)[0][1][2]
```

Note: If device is indicated *and* the AUDIO_TS directory contains a DVDAUDIO.MKB file, unprotection will be performed automatically if supported on the user's platform. Otherwise, the files are assumed to be unprotected.

1.10.1 DVDATitle Objects

class audiotools.**DVDATitle**(*dvdaudio*, *titleset*, *title*, *pts_length*, *tracks*)
This class represents a single DVD-Audio title. dvdaudio is a DVDAudio object. titleset and title are integers indicating this title's position in the DVD-Audio - both offset from 0. pts_length is the the total length of the title in PTS ticks (there are 90000 PTS ticks per second). tracks is a list of DVDATrack objects.

It is rarely instantiated directly; one usually retrieves titles from the parent DVDAudio object.

DVDATitle.**dvdaudio**
The parent DVDAudio object.

DVDATitle.**titleset**
An integer of this title's titleset, offset from 0.

DVDATitle.**title**
An integer of this title's position within the titleset, offset from 0.

DVDATitle.**pts_length**
The length of this title in PTS ticks.

DVDATitle.**tracks**
A list of DVDATrack objects.

DVDATitle.**info**()
Returns a (sample_rate, channels, channel_mask, bits_per_sample, type) tuple of integers. type is 0xA0 if the title is a PCM stream, or 0xA1 if the title is an MLP stream.

DVDATitle.**stream**()
Returns an AOBStream object of this title's data.

DVDATitle.**to_pcm**()
Returns a PCMReader-compatible object of this title's entire data stream.

1.10.2 DVDATrack Objects

class audiotools.**DVDATrack**(*dvdaudio*, *titleset*, *title*, *track*, *first_pts*, *pts_length*, *first_sector*, *last_sector*)
This class represents a single DVD-Audio track. dvdaudio is a DVDAudio object. titleset, title and track are integers indicating this track's position in the DVD-Audio - all offset from 0. first_pts is the track's first PTS value. pts_length is the the total length of the track in PTS ticks. first_sector and last_sector indicate the range of sectors this track occupies.

It is also rarely instantiated directly; one usually retrieves tracks from the parent DVDATitle object.

DVDATrack.**dvdaudio**
The parent DVDAudio object.

DVDATrack.**titleset**
An integer of this tracks's titleset, offset from 0.

DVDATrack.**title**
An integer of this track's position within the titleset, offset from 0.

DVDATrack.**track**
An integer of this track's position within the title, offset from 0.

DVDATrack.**first_pts**
The track's first PTS index.

DVDATrack.**pts_length**
The length of this track in PTS ticks.

DVDATrack.**first_sector**
The first sector this track occupies.

Warning: The track is *not* guaranteed to start at the beginning of its first sector. Although it begins within that sector, the track's start may be offset some arbitrary number of bytes from the sector's start.

DVDATrack.**last_sector**
The last sector this track occupies.

1.10.3 AOBStream Objects

class audiotools.**AOBStream**(*aob_files*, *first_sector*, *last_sector*[, *unprotector*])
This is a stream of DVD-Audio AOB data. It contains several convenience methods to make unpacking that data easier. aob_files is a list of complete AOB file path strings. first_sector and last_sector are integers indicating the stream's range of sectors. unprotector is a function which takes a string of binary sector data and returns an unprotected binary string.

AOBStream.**sectors**()
Iterates over a series of 2048 byte, binary strings of sector data for the entire AOB stream. If unprotector is present, those sectors are returned unprotected.

AOBStream.**packets**()
Iterates over a series of packets by wrapping around the sectors iterator. Each sector contains one or more packets. Packets containing audio data (that is, those with a stream ID of 0xBD) are returned while non-audio packets are discarded.

AOBStream.**packet_payloads**()
Iterates over a series of packet data by wrapping around the packets iterator. The payload is the packet with its ID, CRC and padding removed. Concatenating all of a stream's payloads results in a complete MLP or PCM stream suitable for passing to a decoder.

1.11 ExecQueue Objects

class audiotools.**ExecQueue**
This is a class for executing multiple Python functions in parallel across multiple CPUs.

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

ExecQueue.**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).

class audiotools.**ExecQueue2**
This is a class for executing multiple Python functions in parallel across multiple CPUs and receiving results from those functions.

ExecQueue2.**execute**(*function*, *args*[, *kwargs*])
Queues a Python function, list of arguments and optional dictionary of keyword arguments.

ExecQueue2.**run**([*max_processes*])
Executes all queued Python functions, running max_processes number of functions at a time until the entire queue is empty. Returns an iterator of the returned values of those functions. This operates by forking a new subprocess per function with a pipe between them, executing that function in the child process and then transferring the resulting pickled object back to the parent before performing an unconditional exit.

Queued functions that raise an exception or otherwise exit uncleanly yield None. Likewise, any side effects of the called function have no effect on ExecQueue's caller.

1.12 ExecProgressQueue Objects

class audiotools.**ExecProgressQueue**(*progress_display*)

This class runs multiple jobs in parallel and displays their progress output to the given ProgressDisplay object.

ExecProgressQueue.**results**

A dict of results returned by the queued functions once executed. The key is an integer starting from 0.

Note: Why not a list? Since jobs may finish in an arbitrary order, a dict is used so that results can be accumulated out-of-order. Even using placeholder values such as None may not be appropriate if queued functions return None as a significant value.

ExecProgressQueue.**execute**(*function*[, *progress_text*[, *completion_output*[, **args*[, ***kwargs*]]]])

Queues a Python function for execution. This function is passed the optional args and kwargs arguments upon execution. However, this function is also passed an *additional* progress keyword argument which is a function that takes current and total integer arguments. The executed function can then call that progress function at regular intervals to indicate its progress.

If given, progress_text is a unicode string to be displayed while the function is being executed.

completion_output is displayed once the executed function is completed. It can be either a unicode string or a function whose argument is the returned result of the executed function and which must output a unicode string.

ExecProgressQueue.**run**([*max_processes*])

Executes all the queued 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 in which the running progress and function output are piped to the parent for display to the screen or accumulation in the ExecProgressQueue.results dict.

If an exception occurs in one of the subprocesses, that exception will be raised by ExecProgressQueue.run() and all the running jobs will be terminated.

```
>>> def progress_function(progress, filename):
...   # perform work here
...   progress(current, total)
...   # more work
...   result.a = a
...   result.b = b
...   result.c = c
...   return result
...
>>> def format_result(result):
...    return u"%s %s %s" % (result.a, result.b, result.c)
...
>>> queue = ExecProgressQueue(ProgressDisplay(Messenger("executable")))
>>> queue.execute(function=progress_function,
...               progress_text=u"%s progress" % (filename1),
...               completion_output=format_result,
...               filename=filename1)
...
>>> queue.execute(function=progress_function,
...               progress_text=u"%s progress" % (filename2),
...               completion_output=format_result,
...               filename=filename2)
...
>>> queue.run()
>>> queue.results
```

1.13 Messenger Objects

class audiotools.**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".

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

Messenger.**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".

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

Messenger.**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.

Messenger.**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!
```

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

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

Messenger.**os_error**(*oserror*)
Given an OSError object, displays it as a properly formatted error message with an appended newline.

Note: This is necessary because of the way OSError handles its embedded filename string. Using this method ensures that filename is properly encoded when displayed. Otherwise, there's a good chance that non-ASCII filenames will be garbled.

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

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

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

Messenger.**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.

Messenger.**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.

Messenger.**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.

Messenger.**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
```

Messenger.**info_rows**()
Functions like `Messenger.output_rows()`, but displays output via `Messenger.info()` rather than `Messenger.output()`.

Messenger.**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
```

Messenger.**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.

Messenger.**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

Messenger.**ansi_clearline**()

Generates a ANSI escape codes to clear the current line.

This works only if stdout is a TTY, otherwise is does nothing.

```
>>> msg = Messenger("audiotools", None)
>>> msg.partial_output(u"working")
>>> time.sleep(1)
>>> msg.ansi_clearline()
>>> msg.output(u"done")
```

Messenger.**ansi_uplines**(*self*, *lines*)

Moves the cursor upwards by the given number of lines.

Messenger.**ansi_cleardown**(*self*)

Clears all the output below the current line. This is typically used in conjuction with Messenger.ansi_uplines().

```
>>> msg = Messenger("audiotools", None)
>>> msg.output(u"line 1")
>>> msg.output(u"line 2")
>>> msg.output(u"line 3")
>>> msg.output(u"line 4")
>>> time.sleep(2)
>>> msg.ansi_uplines(4)
>>> msg.ansi_cleardown()
>>> msg.output(u"done")
```

Messenger.**terminal_size**(*fd*)

Given a file descriptor integer, or file object with a fileno() method, returns the size of the current terminal as a (height, width) tuple of integers.

1.14 ProgressDisplay Objects

class audiotools.**ProgressDisplay**(*messenger*)
This is a class for displaying incremental progress updates to the screen. It takes a Messenger object which is used for generating output. Whether or not sys.stdout is a TTY determines how this class operates. If a TTY is detected, screen updates are performed incrementally with individual rows generated and refreshed as needed using ANSI escape sequences such that the user's screen need not scroll. If a TTY is not detected, most progress output is omitted.

ProgressDisplay.**add_row**(*row_id*, *output_line*)
Adds a row of output to be displayed with progress indicated. row_id should be a unique identifier, typically an int. output_line should be a unicode string indicating what we're displaying the progress of.

ProgressDisplay.**update_row**(*row_id*, *current*, *total*)
Updates the progress of the given row. current and total are integers such that current / total indicates the percentage of progress performed.

ProgressDisplay.**refresh**()
Refreshes the screen output, clearing and displaying a fresh progress rows as needed. This is called automatically by update_row().

ProgressDisplay.**clear**()
Clears the screen output. Although refresh() will call this method as needed, one may need to call it manually when generating output independently for the progress monitor so that partial updates aren't left on the user's screen.

ProgressDisplay.**delete_row**(*row_id*)
Removes the row with the given ID from the current list of progress monitors.

class audiotools.**SingleProgressDisplay**(*messenger*, *progress_text*)
This is a subclass of ProgressDisplay used for generating only a single line of progress output. As such, one only specifies a single row of unicode progress_text at initialization time and can avoid the row management functions entirely.

SingleProgressDisplay.**update**(*current*, *total*)
Updates the status of our output row with current and total integers, which function identically to those of ProgressDisplay.update_row().

class audiotools.**ReplayGainProgressDisplay**(*messenger*, *lossless_replay_gain*)
This is another ProgressDisplay subclass optimized for the display of ReplayGain application progress. messenger is a Messenger object and lossless_replay_gain is a boolean indicating whether ReplayGain is being applied losslessly or not (which can be determined from the AudioFile.lossless_replay_gain() classmethod). Whether or not sys.stdout is a TTY determines how this class behaves.

ReplayGainProgressDisplay.**initial_message**()
If operating on a TTY, this does nothing since progress output will be displayed. Otherwise, this indicates that ReplayGain application has begun.

ReplayGainProgressDisplay.**update**(*current*, *total*)
Updates the status of ReplayGain application.

ReplayGainProgressDisplay.**final_message**()
If operating on a TTY, this indicates that ReplayGain application is complete. Otherwise, this does nothing.

```
>>> rg_progress = ReplayGainProgressDisplay(messenger, AudioType.lossless_replay_gain())
>>> rg_progress.initial_message()
>>> AudioType.add_replay_gain(filename_list, rg_progress.update)
>>> rg_Progress.final_message()
```

class audiotools.**ProgressRow**(*row_id*, *output_line*)
This is used by ProgressDisplay and its subclasses for actual output generation. row_id is a unique identifier and output_line is a unicode string. It is not typically instantiated directly.

ProgressRow.**update**(*current*, *total*)
Updates the current progress with `current` and `total` integer values.

ProgressRow.**unicode**(*width*)
Returns the output line and its current progress as a unicode string, formatted to the given width in onscreen characters. Screen width can be determined from the `Messenger.terminal_size()` method.

1.14.1 display_unicode Objects

This class is for displaying portions of a unicode string to the screen. The reason this is needed is because not all Unicode characters are the same width. So, for example, if one wishes to display a portion of a unicode string to a screen that's 80 ASCII characters wide, one can't simply perform:

```
>>> messenger.output(unicode_string[0:80])
```

since some of those Unicode characters might be double width, which would cause the string to wrap.

class audiotools.**display_unicode**(*unicode_string*)

display_unicode.**head**(*display_characters*)
Returns a new `display_unicode` object that's been truncated to the given number of display characters.

```
>>> s = u"".join(map(unichr, range(0x30a1, 0x30a1+25)))
>>> len(s)
25
>>> u = unicode(display_unicode(s).head(40))
>>> len(u)
20
>>> print repr(u)
u'\u30a1\u30a2\u30a3\u30a4\u30a5\u30a6\u30a7\u30a8\u30a9\u30aa\u30ab\u30ac\u30ad\u30ae\u30af\
```

display_unicode.**tail**(*display_characters*)
Returns a new `display_unicode` object that's been truncated to the given number of display characters.

```
>>> s = u"".join(map(unichr, range(0x30a1, 0x30a1+25)))
>>> len(s)
25
>>> u = unicode(display_unicode(s).tail(40))
>>> len(u)
20
>>> print repr(u)
u'\u30a6\u30a7\u30a8\u30a9\u30aa\u30ab\u30ac\u30ad\u30ae\u30af\u30b0\u30b1\u30b2\u30b3\u30b4\
```

display_unicode.**split**(*display_characters*)
Returns a tuple of `display_unicode` objects. The first is up to `display_characters` wide, while the second contains the remainder.

```
>>> s = u"".join(map(unichr, range(0x30a1, 0x30a1+25)))
>>> (head, tail) = display_unicode(s).split(40)
>>> print repr(unicode(head))
u'\u30a1\u30a2\u30a3\u30a4\u30a5\u30a6\u30a7\u30a8\u30a9\u30aa\u30ab\u30ac\u30ad\u30ae\u30af\
>>> print repr(unicode(tail))
u'\u30b5\u30b6\u30b7\u30b8\u30b9'
```

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.

audiotools.pcm.**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]
```

audiotools.pcm.**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]
```

audiotools.pcm.**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]
```

audiotools.pcm.**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]
```

audiotools.pcm.**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 audiotools.pcm.**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.

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

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

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

FrameList.**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.

FrameList.**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.

FrameList.**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.

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

FrameList.**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.

FrameList.**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 audiotools.pcm.**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.

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

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

FloatFrameList.**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.

FloatFrameList.**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.

FloatFrameList.**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.

FloatFrameList.**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 audiotools.resample.**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)
```

Resampler.**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, and the ReplayGainReader class for applying those gains to a audiotools.PCMReader stream.

4.1 ReplayGain Objects

class audiotools.replaygain.**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.

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

ReplayGain.**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().

ReplayGain.**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.

4.2 ReplayGainReader Objects

class audiotools.replaygain.**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.

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 audiotools.cdio.**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.

CDDA.**total_tracks**()

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

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

CDDA.**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)
```

CDDA.**first_sector**()

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

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

CDDA.**last_sector**()

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

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

CDDA.**length_in_seconds**()

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

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

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

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

CDDA.**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)
```

CDDA.**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
```

CDDA.**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
```

audiotools.cdio.**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.

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

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

6.1 Cuesheet Objects

class audiotools.cue.**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.

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

Cuesheet.**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.

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

Cuesheet.**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.

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

classmethod Cuesheet.**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.

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

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

7.1 TOCFile Objects

class audiotools.toc.**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.

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

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

TOCFile.**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.

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

classmethod TOCFile.**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

AUDIOTOOLS.PLAYER — THE AUDIO PLAYER MODULE

The audiotools.player module contains the Player and AudioOutput classes for playing AudioFiles.

audiotools.player.**AUDIO_OUTPUT**
A tuple of AudioOutput-compatible classes of available output types. As with AVAILABLE_TYPES, these are classes that are available to audiotools, not necessarily availble to the user.

Class	Output System
PulseAudioOutput	PulseAudio
OSSAudioOutput	OSS
PortAudioOutput	PortAudio
NULLAudioOutput	No output

8.1 Player Objects

This class is an audio player which plays audio data from an opened audio file object to a given output sink.

class audiotools.player.**Player**(*audio_output*[, *replay_gain*[, *next_track_callback*]])
audio_output is a AudioOutput object subclass which audio data will be played to. replay_gain is either RG_NO_REPLAYGAIN, RG_TRACK_GAIN or RG_ALBUM_GAIN, indicating the level of ReplayGain to apply to tracks being played back. next_track_callback is a function which takes no arguments, to be called when the currently playing track is completed.

Player.**open**(*audiofile*)
Opens the given audiotools.AudioFile object for playing. Any currently playing file is stopped.

Player.**play**()
Begins or resumes playing the currently opened audiotools.AudioFile object, if any.

Player.**set_replay_gain**(*replay_gain*)
Sets the given ReplayGain level to apply during playback. Choose from RG_NO_REPLAYGAIN, RG_TRACK_GAIN or RG_ALBUM_GAIN ReplayGain cannot be applied mid-playback. One must stop() and play() a file for it to take effect.

Player.**pause**()
Pauses playback of the current file. Playback may be resumed with play() or toggle_play_pause()

Player.**toggle_play_pause**()
Pauses the file if playing, play the file if paused.

Player.**stop**()
Stops playback of the current file. If play() is called, playback will start from the beginning.

Player.**close**()
Closes the player for playback. The player thread is halted and the AudioOutput object is closed.

Player.**progress**()
Returns a (pcm_frames_played, pcm_frames_total) tuple. This indicates the current playback status in terms of PCM frames.

8.2 CDPlayer Objects

This class is an audio player which plays audio data from a CDDA disc to a given output sink.

class audiotools.player.**CDPlayer**(*cdda*, *audio_output*[, *next_track_callback*])
cdda is a audiotools.CDDA object. audio_output is a AudioOutput object subclass which audio data will be played to. next_track_callback is a function which takes no arguments, to be called when the currently playing track is completed.

CDPlayer.**open**(*track_number*)
Opens the given track number for reading, where track_number starts from 1.

CDPlayer.**play**()
Begins or resumes playing the currently opened track, if any.

CDPlayer.**pause**()
Pauses playback of the current track. Playback may be resumed with play() or toggle_play_pause()

CDPlayer.**toggle_play_pause**()
Pauses the track if playing, play the track if paused.

CDPlayer.**stop**()
Stops playback of the current track. If play() is called, playback will start from the beginning.

CDPlayer.**close**()
Closes the player for playback. The player thread is halted and the AudioOutput object is closed.

CDPlayer.**progress**()
Returns a (pcm_frames_played, pcm_frames_total) tuple. This indicates the current playback status in terms of PCM frames.

8.3 AudioOutput Objects

This is an abstract class used to implement audio output sinks.

class audiotools.player.**AudioOutput**()

AudioOutput.**NAME**
The name of the AudioOutput subclass as a string.

AudioOutput.**compatible**(*pcmreader*)
Returns True if the given audiotools.PCMReader is compatible with the currently opened output stream. If False, one should call init() in order to reinitialize the output stream to play the given reader.

AudioOutput.**init**(*sample_rate*, *channels*, *channel_mask*, *bits_per_sample*)
Initializes the output stream for playing audio with the given parameters. This *must* be called prior to play() and close().

AudioOutput.**framelist_converter**()
Returns a function which converts audiotools.pcm.FrameList objects to objects which are compatible with our play() method, for the currently initialized stream.

AudioOutput.**play**(*data*)
Plays the converted data object to our output stream.

Note: Why not simply have the `play()` method perform PCM conversion itself instead of shifting it to `framelist_converter()`? The reason is that conversion may be a relatively time-consuming task. By shifting that process into a subthread, there's less chance that performing that work will cause playing to stutter while it completes.

`AudioOutput.`**`close`**`()`
Closes the output stream for further playback.

classmethod `AudioOutput.`**`available`**`()`
Returns True if the AudioOutput implementation is available on the system.

CHAPTER

NINE

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.

9.1 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
```

classmethod `ApeTag.`**`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.

`ApeTag.`**`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).

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

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

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

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

9.2 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:

FlacMetaData.**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.

FlacMetaData.**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().

FlacMetaData.**cuesheet**
A FlacCueSheet object containing CUESHEET data, or None.

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

FlacMetaData.**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().

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

FlacMetaData.**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.

FlacMetaDataBlock.**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.

FlacVorbisComment.**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.

FlacPictureComment.**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.

FlacPictureComment.**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.

FlacCueSheet.**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.

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

9.3 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

ID3v1Comment.**build_tag**()
Returns this tag as a string.

classmethod ID3v1Comment.**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.

classmethod ID3v1Comment.**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.

9.4 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.

`ID3v22Frame.`**build**`()`

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

classmethod `ID3v22Frame.`**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

`ID3v22TextFrame.`**__int__**`()`

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

ID3v22TextFrame.**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
```

classmethod ID3v22TextFrame.**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.

classmethod ID3v22ComFrame.**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

ID3v22PicFrame.**type_string**()
Returns the pic_type as a plain string.

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

9.5 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.

ID3v23Frame.**build**()
Returns the frame's contents as a string of binary data.

classmethod ID3v23Frame.**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

ID3v23TextFrame.**__int__**()
Returns the first integer portion of the frame data as an int.

ID3v23TextFrame.**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
```

classmethod ID3v23TextFrame.**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.

classmethod ID3v23ComFrame.**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

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

9.6 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.

`ID3v24Frame.`**build**`()`

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

classmethod `ID3v24Frame.`**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

ID3v24TextFrame.**__int__**()
Returns the first integer portion of the frame data as an int.

ID3v24TextFrame.**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
```

classmethod ID3v24TextFrame.**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.

classmethod ID3v24ComFrame.**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

classmethod ID3v23PicFrame.**converted**(*image*)
Given an audiotools.Image object, returns a new ID3v24PicFrame object.

9.7 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'
```

ID3CommentPair.**id3v2**

The embedded ID3v22Comment, ID3v23Comment or ID3v24Comment

ID3CommentPair.**id3v1**

The embedded ID3v1Comment

9.8 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.

M4AMetaData.**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.

classmethod M4AMetaData.**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.

classmethod M4AMetaData.**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.

classmethod M4AMetaData.**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.

classmethod M4AMetaData.**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.

classmethod M4AMetaData.**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.

9.9 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'
```

VorbisComment.**build**()

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

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

a

audiotools, 3
audiotools.cdio, 35
audiotools.cue, 37
audiotools.pcm, 27
audiotools.player, 41
audiotools.replaygain, 33
audiotools.resample, 31
audiotools.toc, 39

INDEX

Symbols

__int__() (ID3v22TextFrame method), 49
__int__() (ID3v23TextFrame method), 51
__int__() (ID3v24TextFrame method), 53
__len__() (audiotools.AlbumMetaDataFile method), 11

A

add_image() (audiotools.MetaData method), 11
add_replay_gain() (audiotools.AudioFile class method), 8
add_row() (audiotools.ProgressDisplay method), 25
AiffContainer (class in audiotools), 9
album_gain() (audiotools.replaygain.ReplayGain method), 33
album_number() (audiotools.AudioFile method), 7
AlbumMetaData (class in audiotools), 11
AlbumMetaDataFile (class in audiotools), 11
AlbumMetaDataFile.album_name (in module audiotools), 11
AlbumMetaDataFile.artist_name (in module audiotools), 11
AlbumMetaDataFile.catalog (in module audiotools), 11
AlbumMetaDataFile.extra (in module audiotools), 11
AlbumMetaDataFile.year (in module audiotools), 11
ansi() (audiotools.Messenger method), 23
ansi_cleardown() (audiotools.Messenger method), 24
ansi_clearline() (audiotools.Messenger method), 24
ansi_err() (audiotools.Messenger method), 23
ansi_uplines() (audiotools.Messenger method), 24
AOBStream (class in audiotools), 20
ApeTag (built-in class), 45
ApeTagItem (built-in class), 46
applicable_replay_gain() (in module audiotools), 4
AUDIO_OUTPUT (in module audiotools.player), 41
AudioFile (class in audiotools), 5
AudioOutput (class in audiotools.player), 42
AudioOutput.NAME (in module audiotools.player), 42
audiotools (module), 3
audiotools.cdio (module), 35
audiotools.cue (module), 37
audiotools.pcm (module), 27
audiotools.player (module), 41
audiotools.replaygain (module), 33
audiotools.resample (module), 31
audiotools.toc (module), 39
available() (audiotools.player.AudioOutput class method), 43
AVAILABLE_TYPES (in module audiotools), 3

B

back_covers() (audiotools.MetaData method), 11
BIN (in module audiotools), 3
BINARIES (audiotools.AudioFile attribute), 5
binary() (ApeTagItem class method), 46
binary_atom() (M4AMetaData class method), 56
bits_per_sample() (audiotools.AudioFile method), 6
blank_row() (audiotools.Messenger method), 23
BufferedPCMReader (class in audiotools), 15
build() (ApeTag method), 46
build() (ApeTagItem method), 46
build() (FlacMetaData method), 47
build() (FlacPictureComment method), 47
build() (ID3v22Frame method), 49
build() (ID3v23Frame method), 51
build() (ID3v24Frame method), 53
build() (VorbisComment method), 57
build_block() (FlacCueSheet method), 48
build_block() (FlacMetaDataBlock method), 47
build_block() (FlacPictureComment method), 47
build_block() (FlacVorbisComment method), 47
build_id3v1() (ID3v1Comment class method), 48
build_tag() (ID3v1Comment method), 48

C

calculate_replay_gain() (in module audiotools), 5
can_add_replay_gain() (audiotools.AudioFile class method), 8
catalog() (audiotools.cue.Cuesheet method), 37
catalog() (audiotools.toc.TOCFile method), 39
cd_frames() (audiotools.AudioFile method), 6
CDDA (class in audiotools), 17
CDDA (class in audiotools.cdio), 35
CDPlayer (class in audiotools.player), 42
CDTrackLog (class in audiotools), 18
CDTrackReader (class in audiotools), 18
CDTrackReader.rip_log (in module audiotools), 18
channel() (audiotools.pcm.FloatFrameList method), 29
channel() (audiotools.pcm.FrameList method), 28
channel_mask() (audiotools.AudioFile method), 6
ChannelMask (class in audiotools), 16
channels() (audiotools.AudioFile method), 6

channels() (audiotools.ChannelMask method), 17
clear() (audiotools.ProgressDisplay method), 25
close() (audiotools.PCMConverter method), 14
close() (audiotools.PCMReader method), 14
close() (audiotools.player.AudioOutput method), 43
close() (audiotools.player.CDPlayer method), 42
close() (audiotools.player.Player method), 41
compatible() (audiotools.player.AudioOutput method), 42
COMPRESSION_DESCRIPTIONS (audiotools.AudioFile attribute), 5
COMPRESSION_MODES (audiotools.AudioFile attribute), 5
convert() (audiotools.AudioFile method), 6
converted() (audiotools.MetaData class method), 10
converted() (FlacCueSheet class method), 48
converted() (ID3v22PicFrame class method), 50
converted() (ID3v23PicFrame class method), 52, 54
covr_atom() (M4AMetaData class method), 56
CueException, 37
Cuesheet (class in audiotools.cue), 37

D

DEFAULT_COMPRESSION (audiotools.AudioFile attribute), 5
defined() (audiotools.ChannelMask method), 17
delete_image() (audiotools.MetaData method), 11
delete_metadata() (audiotools.AudioFile method), 6
delete_row() (audiotools.ProgressDisplay method), 25
disk_atom() (M4AMetaData class method), 56
display_unicode (class in audiotools), 26
divider_row() (audiotools.Messenger method), 23
DVDATitle (class in audiotools), 19
DVDATitle.dvdaudio (in module audiotools), 19
DVDATitle.pts_length (in module audiotools), 19
DVDATitle.title (in module audiotools), 19
DVDATitle.titleset (in module audiotools), 19
DVDATitle.tracks (in module audiotools), 19
DVDATrack (class in audiotools), 19
DVDATrack.dvdaudio (in module audiotools), 19
DVDATrack.first_pts (in module audiotools), 19
DVDATrack.first_sector (in module audiotools), 19
DVDATrack.last_sector (in module audiotools), 20
DVDATrack.pts_length (in module audiotools), 19
DVDATrack.title (in module audiotools), 19
DVDATrack.titleset (in module audiotools), 19
DVDATrack.track (in module audiotools), 19
DVDAudio (class in audiotools), 18

E

error() (audiotools.Messenger method), 22
ExecProgressQueue (class in audiotools), 21
ExecQueue (class in audiotools), 20
ExecQueue2 (class in audiotools), 20
execute() (audiotools.ExecProgressQueue method), 21
execute() (audiotools.ExecQueue method), 20
execute() (audiotools.ExecQueue2 method), 20
external() (ApeTagItem class method), 46

F

file() (audiotools.cue.Cuesheet class method), 37
file() (audiotools.toc.TOCFile class method), 39
filename() (audiotools.Messenger method), 22
filename_to_type() (in module audiotools), 4
final_message() (audiotools.ReplayGainProgressDisplay method), 25
first_sector() (audiotools.CDDA method), 18
first_sector() (audiotools.cdio.CDDA method), 35
FlacCueSheet (built-in class), 48
FlacMetaData (built-in class), 46
FlacMetaData.cuesheet (built-in variable), 47
FlacMetaData.extra_blocks (built-in variable), 47
FlacMetaData.image_blocks (built-in variable), 47
FlacMetaData.streaminfo (built-in variable), 47
FlacMetaData.vorbis_comment (built-in variable), 47
FlacMetaDataBlock (built-in class), 47
FlacPictureComment (built-in class), 47
FlacVorbisComment (built-in class), 47
FloatFrameList (class in audiotools.pcm), 29
FloatFrameList.channels (in module audiotools.pcm), 29
FloatFrameList.frames (in module audiotools.pcm), 29
frame() (audiotools.pcm.FloatFrameList method), 29
frame() (audiotools.pcm.FrameList method), 28
frame_count() (audiotools.pcm.FrameList method), 28
FrameList (class in audiotools.pcm), 28
FrameList.bits_per_sample (in module audiotools.pcm), 28
FrameList.channels (in module audiotools.pcm), 28
FrameList.frames (in module audiotools.pcm), 28
framelist_converter() (audiotools.player.AudioOutput method), 42
from_aiff() (audiotools.AiffContainer class method), 9
from_channels() (audiotools.ChannelMask class method), 17
from_channels() (in module audiotools.pcm), 27
from_cuesheet() (audiotools.AlbumMetaDataFile class method), 12
from_fields() (audiotools.ChannelMask class method), 17
from_float_channels() (in module audiotools.pcm), 27
from_float_frames() (in module audiotools.pcm), 27
from_frames() (in module audiotools.pcm), 27
from_list() (in module audiotools.pcm), 27
from_pcm() (audiotools.AudioFile class method), 6
from_string() (audiotools.AlbumMetaDataFile class method), 12
from_tracks() (audiotools.AlbumMetaDataFile class method), 12
from_unicode() (ID3v22ComFrame class method), 50
from_unicode() (ID3v22TextFrame class method), 50
from_unicode() (ID3v23ComFrame class method), 52
from_unicode() (ID3v23TextFrame class method), 52
from_unicode() (ID3v24ComFrame class method), 54
from_unicode() (ID3v24TextFrame class method), 54
from_wave() (audiotools.WaveContainer class method), 9

front_covers() (audiotools.MetaData method), 10

G

get() (audiotools.AlbumMetaDataFile method), 12
get_cuesheet() (audiotools.AudioFile method), 8
get_metadata() (audiotools.AudioFile method), 6
get_track() (audiotools.AlbumMetaDataFile method), 12
group_tracks() (in module audiotools), 4

H

has_binaries() (audiotools.AudioFile class method), 8
has_foreign_aiff_chunks() (audiotools.AiffContainer method), 9
has_foreign_riff_chunks() (audiotools.WaveContainer method), 9
head() (audiotools.display_unicode method), 26

I

ID3CommentPair (built-in class), 55
ID3CommentPair.id3v1 (built-in variable), 55
ID3CommentPair.id3v2 (built-in variable), 55
ID3v1Comment (built-in class), 48
ID3v22ComFrame (built-in class), 50
ID3v22Comment (built-in class), 49
ID3v22Frame (built-in class), 49
ID3v22PicFrame (built-in class), 50
ID3v22TextFrame (built-in class), 49
ID3v23ComFrame (built-in class), 52
ID3v23Comment (built-in class), 51
ID3v23Frame (built-in class), 51
ID3v23PicFrame (built-in class), 52
ID3v23TextFrame (built-in class), 51
ID3v24ComFrame (built-in class), 54
ID3v24Comment (built-in class), 53
ID3v24Frame (built-in class), 53
ID3v24PicFrame (built-in class), 54
ID3v24TextFrame (built-in class), 53
ILST_Atom (built-in class), 56
Image (class in audiotools), 12
Image.color_count (in module audiotools), 12
Image.color_depth (in module audiotools), 12
Image.data (in module audiotools), 12
Image.description (in module audiotools), 13
Image.height (in module audiotools), 12
Image.mime_type (in module audiotools), 12
Image.type (in module audiotools), 13
Image.width (in module audiotools), 12
images() (audiotools.MetaData method), 10
index() (audiotools.ChannelMask method), 17
indexes() (audiotools.cue.Cuesheet method), 37
indexes() (audiotools.toc.TOCFile method), 39
info() (audiotools.DVDATitle method), 19
info() (audiotools.Messenger method), 22
info_rows() (audiotools.Messenger method), 23
init() (audiotools.player.AudioOutput method), 42
initial_message() (audiotools.ReplayGainProgressDisplay method), 25
is_type() (audiotools.AudioFile class method), 5
ISRCs() (audiotools.cue.Cuesheet method), 37
ISRCs() (audiotools.toc.TOCFile method), 39

L

last_sector() (audiotools.CDDA method), 18
last_sector() (audiotools.cdio.CDDA method), 35
leaflet_pages() (audiotools.MetaData method), 11
length() (audiotools.CDDA method), 18
length() (audiotools.CDTrackReader method), 18
length_in_seconds() (audiotools.cdio.CDDA method), 35
LimitedPCMReader (class in audiotools), 15
lossless() (audiotools.AudioFile method), 6
lossless_replay_gain() (audiotools.AudioFile class method), 8

M

M4AMetaData (built-in class), 55
media_images() (audiotools.MetaData method), 11
merge() (audiotools.MetaData method), 11
Messenger (class in audiotools), 22
MetaData (class in audiotools), 9
metadata() (audiotools.AlbumMetaData method), 11
metadata() (audiotools.AlbumMetaDataFile method), 12
MetaData.album_name (in module audiotools), 10
MetaData.album_number (in module audiotools), 10
MetaData.album_total (in module audiotools), 10
MetaData.artist_name (in module audiotools), 10
MetaData.catalog (in module audiotools), 10
MetaData.comment (in module audiotools), 10
MetaData.composer_name (in module audiotools), 10
MetaData.conductor_name (in module audiotools), 10
MetaData.date (in module audiotools), 10
MetaData.ISRC (in module audiotools), 10
MetaData.media (in module audiotools), 10
MetaData.performer_name (in module audiotools), 10
MetaData.track_name (in module audiotools), 9
MetaData.track_number (in module audiotools), 9
MetaData.track_total (in module audiotools), 9
MetaData.year (in module audiotools), 10
metadata_blocks() (FlacMetaData method), 47

N

NAME (audiotools.AudioFile attribute), 5
new() (audiotools.Image class method), 13
new_row() (audiotools.Messenger method), 22

O

offset() (audiotools.CDTrackReader method), 18
open() (audiotools.player.CDPlayer method), 42
open() (audiotools.player.Player method), 41
open() (in module audiotools), 3
open_directory() (in module audiotools), 4

open_files() (in module audiotools), 4
os_error() (audiotools.Messenger method), 22
other_images() (audiotools.MetaData method), 11
output() (audiotools.Messenger method), 22
output_column() (audiotools.Messenger method), 22
output_rows() (audiotools.Messenger method), 23

P

packet_payloads() (audiotools.AOBStream method), 20
packets() (audiotools.AOBStream method), 20
parse() (ID3v22Frame class method), 49
parse() (ID3v23Frame class method), 51
parse() (ID3v24Frame class method), 53
partial_info() (audiotools.Messenger method), 22
partial_output() (audiotools.Messenger method), 22
pause() (audiotools.player.CDPlayer method), 42
pause() (audiotools.player.Player method), 41
pcm_cmp() (in module audiotools), 4
pcm_frame_cmp() (in module audiotools), 4
pcm_lengths() (audiotools.cue.Cuesheet method), 37
pcm_lengths() (audiotools.toc.TOCFile method), 39
pcm_split() (in module audiotools), 4
PCMCat (class in audiotools), 15
PCMConverter (class in audiotools), 14
PCMConverter.bits_per_sample (in module audiotools), 14
PCMConverter.channel_mask (in module audiotools), 14
PCMConverter.channels (in module audiotools), 14
PCMConverter.sample_rate (in module audiotools), 14
PCMReader (class in audiotools), 13
PCMReader.bits_per_sample (in module audiotools), 14
PCMReader.channel_mask (in module audiotools), 14
PCMReader.channels (in module audiotools), 14
PCMReader.sample_rate (in module audiotools), 13
PCMReaderError (class in audiotools), 14
PCMReaderProgress (class in audiotools), 16
PCMReaderWindow (class in audiotools), 15
play() (audiotools.player.AudioOutput method), 42
play() (audiotools.player.CDPlayer method), 42
play() (audiotools.player.Player method), 41
Player (class in audiotools.player), 41
process() (audiotools.resample.Resampler method), 31
progress() (audiotools.player.CDPlayer method), 42
progress() (audiotools.player.Player method), 41
ProgressDisplay (class in audiotools), 25
ProgressRow (class in audiotools), 25

R

read() (ApeTag class method), 46
read() (audiotools.PCMConverter method), 14
read() (audiotools.PCMReader method), 14
read_cuesheet() (in module audiotools.cue), 37
read_id3v1_comment() (ID3v1Comment class method), 48
read_metadata_file() (in module audiotools), 5
read_sector() (audiotools.cdio.CDDA method), 36
read_sectors() (audiotools.cdio.CDDA method), 36
read_sheet() (in module audiotools), 5
read_tocfile() (in module audiotools.toc), 39
refresh() (audiotools.ProgressDisplay method), 25
ReorderedPCMReader (class in audiotools), 15
replay_gain() (audiotools.AudioFile method), 8
ReplayGain (class in audiotools), 13
ReplayGain (class in audiotools.replaygain), 33
ReplayGain.album_gain (in module audiotools), 13
ReplayGain.album_peak (in module audiotools), 13
ReplayGain.track_gain (in module audiotools), 13
ReplayGain.track_peak (in module audiotools), 13
REPLAYGAIN_BINARIES (audiotools.AudioFile attribute), 5
ReplayGainProgressDisplay (class in audiotools), 25
ReplayGainReader (class in audiotools.replaygain), 33
Resampler (class in audiotools.resample), 31
results (audiotools.ExecProgressQueue attribute), 21
run() (audiotools.ExecProgressQueue method), 21
run() (audiotools.ExecQueue method), 20
run() (audiotools.ExecQueue2 method), 20

S

sample_rate() (audiotools.AudioFile method), 6
seconds_length() (audiotools.AudioFile method), 6
sectors() (audiotools.AOBStream method), 20
seek() (audiotools.cdio.CDDA method), 36
set_cuesheet() (audiotools.AudioFile method), 8
set_metadata() (audiotools.AudioFile method), 6
set_read_callback() (in module audiotools.cdio), 36
set_replay_gain() (audiotools.player.Player method), 41
set_speed() (audiotools.cdio.CDDA method), 36
set_track() (audiotools.AlbumMetaDataFile method), 12
single_file_type() (audiotools.cue.Cuesheet method), 37
SingleProgressDisplay (class in audiotools), 25
split() (audiotools.display_unicode method), 26
split() (audiotools.pcm.FloatFrameList method), 29
split() (audiotools.pcm.FrameList method), 28
stop() (audiotools.player.CDPlayer method), 42
stop() (audiotools.player.Player method), 41
stream() (audiotools.DVDATitle method), 19
string() (ApeTagItem class method), 46
stripped_pcm_cmp() (in module audiotools), 4
SUFFIX (audiotools.AudioFile attribute), 5
suffix() (audiotools.Image method), 13
supports_images() (audiotools.MetaData class method), 10

T

tail() (audiotools.display_unicode method), 26
terminal_size() (audiotools.Messenger method), 24
text_atom() (M4AMetaData class method), 56
thumbnail() (audiotools.Image method), 13
title_gain() (audiotools.replaygain.ReplayGain method), 33

to_aiff() (audiotools.AiffContainer method), 9
to_atom() (M4AMetaData method), 55
to_bytes() (audiotools.pcm.FrameList method), 28
to_float() (audiotools.pcm.FrameList method), 28
to_int() (audiotools.pcm.FloatFrameList method), 29
to_pcm() (audiotools.AudioFile method), 6
to_pcm() (audiotools.DVDATitle method), 19
to_pcm_progress() (in module audiotools), 5
to_string() (audiotools.AlbumMetaDataFile method), 12
to_wave() (audiotools.WaveContainer method), 9
TOCException, 39
TOCFile (class in audiotools.toc), 39
toggle_play_pause() (audiotools.player.CDPlayer method), 42
toggle_play_pause() (audiotools.player.Player method), 41
total() (ID3v22TextFrame method), 49
total() (ID3v23TextFrame method), 52
total() (ID3v24TextFrame method), 54
total_frames() (audiotools.AudioFile method), 6
total_tracks() (audiotools.cdio.CDDA method), 35
track_metadata() (audiotools.AlbumMetaDataFile method), 12
track_metadatas() (audiotools.AlbumMetaDataFile method), 12
track_name() (audiotools.AudioFile class method), 7
track_number() (audiotools.AudioFile method), 7
track_offsets() (audiotools.cdio.CDDA method), 35
track_type() (audiotools.cdio.CDDA method), 36
transfer_data() (in module audiotools), 4
transfer_framelist_data() (in module audiotools), 4
trkn_atom() (M4AMetaData class method), 56
TYPE_MAP (in module audiotools), 3
type_string() (audiotools.Image method), 13
type_string() (ID3v22PicFrame method), 50

U

undefined() (audiotools.ChannelMask method), 17
unicode() (audiotools.ProgressRow method), 26
update() (audiotools.ProgressRow method), 25
update() (audiotools.replaygain.Replaygain method), 33
update() (audiotools.ReplayGainProgressDisplay method), 25
update() (audiotools.SingleProgressDisplay method), 25
update_row() (audiotools.ProgressDisplay method), 25
usage() (audiotools.Messenger method), 22

V

verify() (audiotools.AudioFile method), 7
VERSION (in module audiotools), 3
VorbisComment (built-in class), 56

W

warning() (audiotools.Messenger method), 22
WaveContainer (class in audiotools), 8