Docs » PySoundFile

PySoundFile

PySoundFile is an audio library based on libsndfile, CFFI and NumPy. Full documentation is available on http://pysoundfile.readthedocs.org/.

PySoundFile can read and write sound files. File reading/writing is supported through libsndfile, which is a free, cross-platform, open-source (LGPL) library for reading and writing many different sampled sound file formats that runs on many platforms including Windows, OS X, and Unix. It is accessed through CFFI, which is a foreign function interface for Python calling C code. CFFI is supported for CPython 2.6+, 3.x and PyPy 2.0+. PySoundFile represents audio data as NumPy arrays.

Breaking Changes

PySoundFile has evolved rapidly during the last few releases. Most notably, we changed the import name from `import pysoundfile` to `import soundfile` in 0.7. In 0.6, we cleaned up many small inconsistencies, particularly in the the ordering and naming of function arguments and the removal of the indexing interface.

In 0.8.0, we changed the default value of `always_2d` from `True` to `False`. Also, the order of arguments of the `write` function changed from `write(data, file, ...)` to `write(file, data, ...)`.

In 0.9.0, we changed the `ctype` arguments of the `buffer_*` methods to `dtype`, using the Numpy `dtype` notation. The old `ctype` arguments still work, but are now officially deprecated.

Installation

PySoundFile depends on the Python packages CFFI and NumPy, and the system library libsndfile.

To install the Python dependencies, I recommend using the Anaconda distribution of Python 3. This will come with all dependencies pre-installed. To install the dependencies manually, you can use the `conda` package manager, which will install all dependencies using `conda install cffi numpy` (conda is also available independently of Anaconda with `pip install conda; conda init`).

With CFFI and NumPy installed, you can use `pip install pysoundfile` to download and install the

latest release of PySoundFile. On Windows and OS X, this will also install the library libsndfile. On Linux, you need to install libsndfile using your distribution's package manager, for example `sudo apt-get install libsndfile1`.

Read/Write Functions

Data can be written to the file using **`soundfile.write()`**, or read from the file using **`soundfile.read()`**. PySoundFile can open all file formats that libsndfile supports, for example WAV, FLAC, OGG and MAT files.

Here is an example for a program that reads a wave file and copies it into an ogg-vorbis file:

```
import soundfile as sf

data, samplerate = sf.read('existing_file.wav')
sf.write('new_file.ogg', data, samplerate)
```

Block Processing

Sound files can also be read in short, optionally overlapping blocks with **`soundfile.blocks()`**. For example, this calculates the signal level for each block of a long file:

```
import numpy as np
import soundfile as sf

rms = [np.sqrt(np.mean(block**2)) for block in
       sf.blocks('myfile.wav', blocksize=1024, overlap=512)]
```

SoundFile Objects

Sound files can also be opened as **`soundfile.SoundFile`** objects. Every SoundFile has a specific sample rate, data format and a set number of channels.

If a file is opened, it is kept open for as long as the SoundFile object exists. The file closes when the object is garbage collected, but you should use the **`soundfile.SoundFile.close()`** method or the context manager to close the file explicitly:

```
import soundfile as sf

with sf.SoundFile('myfile.wav', 'rw') as f:
    while f.tell() < len(f):
        pos = f.tell()
        data = f.read(1024)
        f.seek(pos)
        f.write(data*2)
```

All data access uses frames as index. A frame is one discrete time step in the sound file. Every frame contains as many samples as there are channels in the file.

RAW Files

Pysoundfile can usually auto-detect the file type of sound files. This is not possible for RAW files, though:

```
import soundfile as sf

data, samplerate = sf.read('myfile.raw', channels=1, samplerate=44100,
                           subtype='FLOAT')
```

Note that on x86, this defaults to `endian='LITTLE'`. If you are reading big endian data (mostly old PowerPC/6800-based files), you have to set `endian='BIG'` accordingly.

You can write RAW files in a similar way, but be advised that in most cases, a more expressive format is better and should be used instead.

Virtual IO

If you have an open file-like object, Pysoundfile can open it just like regular files:

```
import soundfile as sf
with open('filename.flac', 'rb') as f:
    data, samplerate = sf.read(f)
```

Here is an example using an HTTP request:

```
import io
import soundfile as sf
from urllib.request import urlopen

url = "http://tinyurl.com/shepard-risset"
data, samplerate = sf.read(io.BytesIO(urlopen(url).read()))
```

Note that the above example only works with Python 3.x. For Python 2.x support, replace the third line with:

```
from urllib2 import urlopen
```

News

2013-08-27 V0.1.0 Bastian Bechtold:

2013-08-30 V0.2.0 Bastian Bechtold:
Bugfixes and more consistency with PySoundCard

2013-08-30 V0.2.1 Bastian Bechtold:
Bugfixes

2013-09-27 V0.3.0 Bastian Bechtold:
Added binary installer for Windows, and context manager

2013-11-06 V0.3.1 Bastian Bechtold:
Switched from distutils to setuptools for easier installation

2013-11-29 V0.4.0 Bastian Bechtold:
Thanks to David Blewett, now with Virtual IO!

2013-12-08 V0.4.1 Bastian Bechtold:
Thanks to Xidorn Quan, FLAC files are not float32 any more.

2014-02-26 V0.5.0 Bastian Bechtold:
Thanks to Matthias Geier, improved seeking and a flush() method.

2015-01-19 V0.6.0 Bastian Bechtold:
A big, big thank you to Matthias Geier, who did most of the work!

- Switched to `float64` as default data type.
- Function arguments changed for consistency.
- Added unit tests.
- Added global `read()`, `write()`, `blocks()` convenience functions.
- Documentation overhaul and hosting on readthedocs.
- Added `'x'` open mode.
- Added `tell()` method.
- Added `__repr__()` method.

2015-04-12 V0.7.0 Bastian Bechtold:
Again, thanks to Matthias Geier for all of his hard work, but also Nils Werner and Whistler7 for their many suggestions and help.

- Renamed `import pysoundfile` to `import soundfile`.
- Installation through pip wheels that contain the necessary libraries for OS X and Windows.
- Removed `exclusive_creation` argument to `write`.
- Added `truncate()` method.

2015-10-20 V0.8.0 Bastian Bechtold:
Again, Matthias Geier contributed a whole lot of hard work to this release.

- Changed the default value of `always_2d` from `True` to `False`.
- Numpy is now optional, and only loaded for `read` and `write`.
- Added `SoundFile.buffer_read` and `SoundFile.buffer_read_into` and `SoundFile.buffer_write`, which read/write raw data without involving Numpy.
- Added `info` function that returns metadata of a sound file.
- Changed the argument order of the `write` function from `write(data, file, ...)` to

And many more minor bug fixes.

2017-02-02 V0.9.0 Bastian Bechtold:

Thank you, Matthias Geier, Tomas Garcia, and Todd, for contributions for this release.

- Adds support for ALAC files.
- Adds new member `__libsndfile_version__`
- Adds number of frames to `info` class
- Adds `dtype` argument to `buffer_*` methods
- Deprecates `ctype` argument to `buffer_*` methods
- Adds official support for Python 3.6

And some minor bug fixes.

Contributing

If you find bugs, errors, omissions or other things that need improvement, please create an issue or a pull request at https://github.com/bastibe/PySoundFile/. Contributions are always welcome!

Testing

If you fix a bug, you should add a test that exposes the bug (to avoid future regressions), if you add a feature, you should add tests for it as well.

To run the tests, use:

```
python setup.py test
```

This uses py.test; if you haven't installed it already, it will be downloaded and installed for you.

Note

There is a known problem that prohibits the use of file descriptors on Windows if the libsndfile DLL was compiled with a different compiler than the Python interpreter. Unfortunately, this is typically the case if the packaged DLLs are used. To skip the tests which utilize file descriptors, use:

```
python setup.py test --pytest-args="-knot\ fd"
```

Coverage

If you want to measure code coverage, you can use coverage.py. Just install it with:

```
pip install coverage --user
```

... and run it with:

```
coverage run --source soundfile.py -m py.test
coverage html
```

The resulting HTML files will be written to the `htmlcov/` directory.

You can even check branch coverage:

```
coverage run --branch --source soundfile.py -m py.test
coverage html
```

Documentation

If you make changes to the documentation, you can re-create the HTML pages on your local system using Sphinx.

You can install it and a few other necessary packages with:

```
pip install -r doc/requirements.txt --user
```

To create the HTML pages, use:

```
python setup.py build_sphinx
```

The generated files will be available in the directory `build/sphinx/html/`.

API Documentation

PySoundFile is an audio library based on libsndfile, CFFI and NumPy.

Sound files can be read or written directly using the functions **`read()`** and **`write()`**. To read a sound

file in a block-wise fashion, use `blocks()`. Alternatively, sound files can be opened as `SoundFile` objects.

For further information, see http://pysoundfile.readthedocs.org/.

`soundfile.read`(*file, frames=-1, start=0, stop=None, dtype='float64', always_2d=False, fill_value=None, out=None, samplerate=None, channels=None, format=None, subtype=None, endian=None, closefd=True*)
[source]

Provide audio data from a sound file as NumPy array.

By default, the whole file is read from the beginning, but the position to start reading can be specified with *start* and the number of frames to read can be specified with *frames*. Alternatively, a range can be specified with *start* and *stop*.

If there is less data left in the file than requested, the rest of the frames are filled with *fill_value*. If no *fill_value* is specified, a smaller array is returned.

Parameters:

- **file** (*str or int or file-like object*) – The file to read from. See `SoundFile` for details.
- **frames** (*int, optional*) – The number of frames to read. If *frames* is negative, the whole rest of the file is read. Not allowed if *stop* is given.
- **start** (*int, optional*) – Where to start reading. A negative value counts from the end.
- **stop** (*int, optional*) – The index after the last frame to be read. A negative value counts from the end. Not allowed if *frames* is given.
- **dtype** (*{'float64', 'float32', 'int32', 'int16'}, optional*) –
  Data type of the returned array, by default `'float64'`. Floating point audio data is typically in the range from `-1.0` to `1.0`. Integer data is in the range from `-2**15` to `2**15-1` for `'int16'` and from `-2**31` to `2**31-1` for `'int32'`.

  Note

  Reading int values from a float file will *not* scale the data to [-1.0, 1.0). If the file contains `np.array([42.6], dtype='float32')`, you will read `np.array([43], dtype='int32')` for `dtype='int32'`.

Returns:

- **audiodata** (*numpy.ndarray or type(out)*) – A two-dimensional NumPy array is returned, where the channels are stored along the first dimension, i.e. as columns. If the sound file has only one channel, a one-dimensional array is returned. Use `always_2d=True` to return a two-dimensional array anyway.

  If *out* was specified, it is returned. If *out* has more frames than available in the file (or if *frames* is smaller than the length of *out*) and no *fill_value* is given, then only a part of *out* is overwritten and a view containing all valid frames is returned.
- **samplerate** (*int*) – The sample rate of the audio file.

Other Parameters:

- **always_2d** (*bool, optional*) – By default, reading a mono sound file will return a one-dimensional array. With `always_2d=True`, audio data is always returned as a two-dimensional array, even if the audio file has only one channel.
- **fill_value** (*float, optional*) – If more frames are requested than available in the file, the rest of the output is be filled with *fill_value*. If *fill_value* is not specified, a smaller array is returned.
- **out** (*numpy.ndarray or subclass, optional*) – If *out* is specified, the data is written into the given array instead of creating a new array. In this case, the arguments *dtype* and *always_2d* are silently ignored! If *frames* is not given, it is obtained from the length of *out*.
- **samplerate, channels, format, subtype, endian, closefd** – See `SoundFile`.

Examples

```
>>> import soundfile as sf
>>> data, samplerate = sf.read('stereo_file.wav')
>>> data
array([[ 0.71329652,  0.06294799],
       [-0.26450912, -0.38874483],
       ...
       [ 0.67398441, -0.11516333]])
>>> samplerate
44100
```

---

soundfile.write(*file, data, samplerate, subtype=None, endian=None, format=None, closefd=True*)
[source]

Write data to a sound file.

Note

If *file* exists, it will be truncated and overwritten!

Parameters:

- **file** (*str or int or file-like object*) – The file to write to. See `SoundFile` for details.
- **data** (*array_like*) –
  The data to write. Usually two-dimensional (channels x frames), but one-dimensional *data* can be used for mono files. Only the data types `'float64'`, `'float32'`, `'int32'` and `'int16'` are supported.

  Note

  The data type of *data* does **not** select the data type of the written file. Audio data will be converted to the given *subtype*. Writing int values to a float file will *not* scale the values to [-1.0, 1.0). If you write the value `np.array([42], dtype='int32')`, to a `subtype='FLOAT'` file, the file will then contain `np.array([42.], dtype='float32')`.

- **samplerate** (*int*) – The sample rate of the audio data.
- **subtype** (*str, optional*) – See `default_subtype()` for the default value and `available_subtypes()` for all possible values.

Other Parameters:

format, endian, closefd – See `SoundFile`.

Examples

Write 10 frames of random data to a new file:

```
>>> import numpy as np
>>> import soundfile as sf
>>> sf.write('stereo_file.wav', np.random.randn(10, 2), 44100, 'PCM_24')
```

---

soundfile.blocks(*file, blocksize=None, overlap=0, frames=-1, start=0, stop=None, dtype='float64', always_2d=False, fill_value=None, out=None, samplerate=None, channels=None, format=None, subtype=None, endian=None, closefd=True*) [source]

Return a generator for block-wise reading.

By default, iteration starts at the beginning and stops at the end of the file. Use *start* to start at a later position and *frames* or *stop* to stop earlier.

If you stop iterating over the generator before it's exhausted, the sound file is not closed. This is normally not a problem because the file is opened in read-only mode. To close the file properly, the generator's `close()` method can be called.

Parameters:

- **file** (*str or int or file-like object*) – The file to read from. See `SoundFile` for details.
- **blocksize** (*int*) – The number of frames to read per block. Either this or *out* must be given.
- **overlap** (*int, optional*) – The number of frames to rewind between each block.

Yields: *numpy.ndarray or type(out)* – Blocks of audio data. If *out* was given, and the requested frames are not an integer multiple of the length of *out*, and no *fill_value* was given, the last block will be a smaller view into *out*.

Other Parameters:

- **frames, start, stop** – See `read()`.
- **dtype** (*{'float64', 'float32', 'int32', 'int16'}, optional*) – See `read()`.
- **always_2d, fill_value, out** – See `read()`.
- **samplerate, channels, format, subtype, endian, closefd** – See `SoundFile`.

Examples

```
>>> import soundfile as sf
>>> for block in sf.blocks('stereo_file.wav', blocksize=1024):
>>>     pass  # do something with 'block'
```

soundfile.info(*file, verbose=False*) [source]

Returns an object with information about a SoundFile.

Parameters: **verbose** (*bool*) – Whether to print additional information.

soundfile.available_formats() [source]

Return a dictionary of available major formats.

Examples

```
>>> import soundfile as sf
>>> sf.available_formats()
{'FLAC': 'FLAC (FLAC Lossless Audio Codec)',
 'OGG': 'OGG (OGG Container format)',
 'WAV': 'WAV (Microsoft)',
 'AIFF': 'AIFF (Apple/SGI)',
 ...
 'WAVEX': 'WAVEX (Microsoft)',
 'RAW': 'RAW (header-less)',
 'MAT5': 'MAT5 (GNU Octave 2.1 / Matlab 5.0)'}
```

soundfile.available_subtypes(*format=None*) [source]

Return a dictionary of available subtypes.

Parameters: **format** (*str*) – If given, only compatible subtypes are returned.

Examples

```
>>> import soundfile as sf
>>> sf.available_subtypes('FLAC')
{'PCM_24': 'Signed 24 bit PCM',
 'PCM_16': 'Signed 16 bit PCM',
 'PCM_S8': 'Signed 8 bit PCM'}
```

soundfile.check_format(*format, subtype=None, endian=None*) [source]

Check if the combination of format/subtype/endian is valid.

Examples

```
>>> import soundfile as sf
>>> sf.check_format('WAV', 'PCM_24')
True
>>> sf.check_format('FLAC', 'VORBIS')
False
```

soundfile.default_subtype(*format*) [source]

Return the default subtype for a given format.

Examples

```
>>> import soundfile as sf
>>> sf.default_subtype('WAV')
'PCM_16'
>>> sf.default_subtype('MAT5')
'DOUBLE'
```

class **soundfile.SoundFile**(*file, mode='r', samplerate=None, channels=None, subtype=None, endian=None, format=None, closefd=True*) [source]

Open a sound file.

If a file is opened with *mode* `'r'` (the default) or `'r+'`, no sample rate, channels or file format need to be given because the information is obtained from the file. An exception is the `'RAW'` data format, which always requires these data points.

File formats consist of three case-insensitive strings:

- a *major format* which is by default obtained from the extension of the file name (if known) and which can be forced with the format argument (e.g. `format='WAVEX'`).
- a *subtype*, e.g. `'PCM_24'`. Most major formats have a default subtype which is used if no subtype is specified.
- an *endian-ness*, which doesn't have to be specified at all in most cases.

A `SoundFile` object is a *context manager*, which means if used in a "with" statement, `close()` is

automatically called when reaching the end of the code block inside the "with" statement.

Parameters:

- **file** (*str or int or file-like object*) – The file to open. This can be a file name, a file descriptor or a Python file object (or a similar object with the methods `read()` / `readinto()`, `write()`, `seek()` and `tell()`).
- **mode** (*{'r', 'r+', 'w', 'w+', 'x', 'x+'}, optional*) – Open mode. Has to begin with one of these three characters: `'r'` for reading, `'w'` for writing (truncates *file*) or `'x'` for writing (raises an error if *file* already exists). Additionally, it may contain `'+'` to open *file* for both reading and writing. The character `'b'` for *binary mode* is implied because all sound files have to be opened in this mode. If *file* is a file descriptor or a file-like object, `'w'` doesn't truncate and `'x'` doesn't raise an error.
- **samplerate** (*int*) – The sample rate of the file. If *mode* contains `'r'`, this is obtained from the file (except for `'RAW'` files).
- **channels** (*int*) – The number of channels of the file. If *mode* contains `'r'`, this is obtained from the file (except for `'RAW'` files).
- **subtype** (*str, sometimes optional*) – The subtype of the sound file. If *mode* contains `'r'`, this is obtained from the file (except for `'RAW'` files), if not, the default value depends on the selected *format* (see `default_subtype()`). See `available_subtypes()` for all possible subtypes for a given *format*.
- **endian** (*{'FILE', 'LITTLE', 'BIG', 'CPU'}, sometimes optional*) – The endian-ness of the sound file. If *mode* contains `'r'`, this is obtained from the file (except for `'RAW'` files), if not, the default value is `'FILE'`, which is correct in most cases.
- **format** (*str, sometimes optional*) – The major format of the sound file. If *mode* contains `'r'`, this is obtained from the file (except for `'RAW'` files), if not, the default value is determined from the file extension. See `available_formats()` for all possible values.
- **closefd** (*bool, optional*) – Whether to close the file descriptor on `close()`. Only applicable if the *file* argument is a file descriptor.

Examples

```
>>> from soundfile import SoundFile
```

Open an existing file for reading:

```
>>> myfile = SoundFile('existing_file.wav')
>>> # do something with myfile
>>> myfile.close()
```

Create a new sound file for reading and writing using a with statement:

```
>>> with SoundFile('new_file.wav', 'x+', 44100, 2) as myfile:
>>>     # do something with myfile
>>>     # ...
>>>     assert not myfile.closed
>>>     # myfile.close() is called automatically at the end
>>> assert myfile.closed
```

name

The file name of the sound file.

mode

The open mode the sound file was opened with.

samplerate

The sample rate of the sound file.

channels

The number of channels in the sound file.

format

The major format of the sound file.

subtype

The subtype of data in the the sound file.

endian

The endian-ness of the data in the sound file.

format_info

A description of the major format of the sound file.

subtype_info

A description of the subtype of the sound file.

sections

The number of sections of the sound file.

closed

Whether the sound file is closed or not.

extra_info

Retrieve the log string generated when opening the file.

seekable() [source]

Return True if the file supports seeking.

seek(*frames*, *whence=0*) [source]

Set the read/write position.

Parameters:
- **frames** (*int*) – The frame index or offset to seek.
- **whence** (*{SEEK_SET, SEEK_CUR, SEEK_END}, optional*) – By default (`whence=SEEK_SET`), *frames* are counted from the beginning of the file. `whence=SEEK_CUR` seeks from the current position (positive and negative values are allowed for *frames*). `whence=SEEK_END` seeks from the end (use negative value for *frames*).

Returns: *int* – The new absolute read/write position in frames.

Examples

```
>>> from soundfile import SoundFile, SEEK_END
>>> myfile = SoundFile('stereo_file.wav')
```

Seek to the beginning of the file:

```
>>> myfile.seek(0)
0
```

Seek to the end of the file:

```
>>> myfile.seek(0, SEEK_END)
44100  # this is the file length
```

tell() [source]

Return the current read/write position.

read(*frames=-1*, *dtype='float64'*, *always_2d=False*, *fill_value=None*, *out=None*) [source]

Read from the file and return data as NumPy array.

Reads the given number of frames in the given data format starting at the current read/write

position. This advances the read/write position by the same number of frames. By default, all frames from the current read/write position to the end of the file are returned. Use `seek()` to move the current read/write position.

Parameters:

- **frames** (*int, optional*) – The number of frames to read. If `frames < 0`, the whole rest of the file is read.
- **dtype** (*{'float64', 'float32', 'int32', 'int16'}, optional*) –
  Data type of the returned array, by default `'float64'`. Floating point audio data is typically in the range from `-1.0` to `1.0`. Integer data is in the range from `-2**15` to `2**15-1` for `'int16'` and from `-2**31` to `2**31-1` for `'int32'`.

  Note

  Reading int values from a float file will *not* scale the data to [-1.0, 1.0). If the file contains `np.array([42.6], dtype='float32')`, you will read `np.array([43], dtype='int32')` for `dtype='int32'`.

Returns:

audiodata (*numpy.ndarray or type(out)*) – A two-dimensional NumPy array is returned, where the channels are stored along the first dimension, i.e. as columns. If the sound file has only one channel, a one-dimensional array is returned. Use `always_2d=True` to return a two-dimensional array anyway.

If *out* was specified, it is returned. If *out* has more frames than available in the file (or if *frames* is smaller than the length of *out*) and no *fill_value* is given, then only a part of *out* is overwritten and a view containing all valid frames is returned. numpy.ndarray or type(out)

Other Parameters:

- **always_2d** (*bool, optional*) – By default, reading a mono sound file will return a one-dimensional array. With `always_2d=True`, audio data is always returned as a two-dimensional array, even if the audio file has only one channel.
- **fill_value** (*float, optional*) – If more frames are requested than available in the file, the rest of the output is be filled with *fill_value*. If *fill_value* is not specified, a smaller array is returned.
- **out** (*numpy.ndarray or subclass, optional*) – If *out* is specified, the data is written into the given array instead of creating a new array. In this case, the arguments *dtype* and *always_2d* are silently ignored! If *frames* is not given, it is obtained from the length of *out*.

Examples

```
>>> from soundfile import SoundFile
>>> myfile = SoundFile('stereo_file.wav')
```

Reading 3 frames from a stereo file:

```
>>> myfile.read(3)
array([[ 0.71329652,  0.06294799],
       [-0.26450912, -0.38874483],
       [ 0.67398441, -0.11516333]])
>>> myfile.close()
```

See also

`buffer_read()`, `write()`

`buffer_read`(*frames=-1*, *ctype=None*, *dtype=None*) [source]

Read from the file and return data as buffer object.

Reads the given number of *frames* in the given data format starting at the current read/write position. This advances the read/write position by the same number of frames. By default, all frames from the current read/write position to the end of the file are returned. Use `seek()` to move the current read/write position.

Parameters:
- **frames** (*int, optional*) – The number of frames to read. If *frames < 0*, the whole rest of the file is read.
- **dtype** (*{'float64', 'float32', 'int32', 'int16'}*) – Audio data will be converted to the given data type.

Returns: *buffer* – A buffer containing the read data.

See also

`buffer_read_into()`, `read()`, `buffer_write()`

`buffer_read_into`(*buffer*, *ctype=None*, *dtype=None*) [source]

Read from the file into a given buffer object.

Fills the given *buffer* with frames in the given data format starting at the current read/write position (which can be changed with `seek()`) until the buffer is full or the end of the file is reached. This advances the read/write position by the number of frames that were read.

Parameters:
- **buffer** (*writable buffer*) – Audio frames from the file are written to this buffer.
- **dtype** (*{'float64', 'float32', 'int32', 'int16'}*) – The data type of *buffer*.

Returns: *int* – The number of frames that were read from the file. This can be less than the size of *buffer*. The rest of the buffer is not filled with meaningful data.

See also

`buffer_read()`, `read()`

`write`(*data*) [source]

Write audio data from a NumPy array to the file.

Writes a number of frames at the read/write position to the file. This also advances the read/write position by the same number of frames and enlarges the file if necessary.

Note that writing int values to a float file will *not* scale the values to [-1.0, 1.0). If you write the value `np.array([42], dtype='int32')`, to a `subtype='FLOAT'` file, the file will then contain `np.array([42.], dtype='float32')`.

Parameters: **data** (*array_like*) –
The data to write. Usually two-dimensional (channels x frames), but one-dimensional *data* can be used for mono files. Only the data types `'float64'`, `'float32'`, `'int32'` and `'int16'` are supported.

Note

The data type of *data* does **not** select the data type of the written file. Audio data will be converted to the given *subtype*. Writing int values to a float file will *not* scale the values to [-1.0, 1.0). If you write the value `np.array([42], dtype='int32')`, to a `subtype='FLOAT'` file, the file will then contain `np.array([42.], dtype='float32')`.

Examples

```
>>> import numpy as np
>>> from soundfile import SoundFile
>>> myfile = SoundFile('stereo_file.wav')
```

Write 10 frames of random data to a new file:

```
>>> with SoundFile('stereo_file.wav', 'w', 44100, 2, 'PCM_24') as f:
>>>     f.write(np.random.randn(10, 2))
```

See also

`buffer_write()`, `read()`

buffer_write(*data*, *ctype=None*, *dtype=None*) [source]

Write audio data from a buffer/bytes object to the file.

Writes the contents of *data* to the file at the current read/write position. This also advances the read/write position by the number of frames that were written and enlarges the file if necessary.

Parameters:
- **data** (*buffer or bytes*) – A buffer or bytes object containing the audio data to be written.
- **dtype** (*{'float64', 'float32', 'int32', 'int16'}*) – The data type of the audio data stored in *data*.

See also

`write()`, `buffer_read()`

blocks(*blocksize=None*, *overlap=0*, *frames=-1*, *dtype='float64'*, *always_2d=False*, *fill_value=None*, *out=None*) [source]

Return a generator for block-wise reading.

By default, the generator yields blocks of the given *blocksize* (using a given *overlap*) until the end of the file is reached; *frames* can be used to stop earlier.

Parameters:
- **blocksize** (*int*) – The number of frames to read per block. Either this or *out* must be given.
- **overlap** (*int, optional*) – The number of frames to rewind between each block.
- **frames** (*int, optional*) – The number of frames to read. If `frames < 1`, the file is read until the end.
- **dtype** (*{'float64', 'float32', 'int32', 'int16'}, optional*) – See `read()`.

Yields: *numpy.ndarray or type(out)* – Blocks of audio data. If *out* was given, and the requested frames are not an integer multiple of the length of *out*, and no *fill_value* was given, the last block will be a smaller view into *out*.

Other Parameters:

always_2d, fill_value, out – See `read()`.

Examples

```
>>> from soundfile import SoundFile
>>> with SoundFile('stereo_file.wav') as f:
>>>     for block in f.blocks(blocksize=1024):
>>>         pass  # do something with 'block'
```

truncate(*frames=None*) [source]

Truncate the file to a given number of frames.

After this command, the read/write position will be at the new end of the file.

Parameters: **frames** (*int, optional*) – Only the data before *frames* is kept, the rest is deleted. If not specified, the current read/write position is used.

flush() [source]

Write unwritten data to the file system.

Data written with `write()` is not immediately written to the file system but buffered in memory to be written at a later time. Calling `flush()` makes sure that all changes are actually written to the file system.

This has no effect on files opened in read-only mode.

close() [source]

Close the file. Can be called multiple times.

Index

- Index