python-lz4 Documentation

Release 4.3.2

Jonathan Underwood

Dec 30, 2022

Contents

CHAPTER 1

Contents

1.1 Introduction

This package provides a Python interface for the LZ4 compression library by Yann Collet. Support is provided for Python 2 (from 2.7 onwards) and Python 3 (from 3.4 onwards).

The LZ4 library provides support for three specifications:

- The frame format
- The block format
- The stream format

This Python interface currently supports the frame, block and double-buffer stream formats.

For most applications, the frame format is what you should use as this guarantees interoperability with other bindings. The frame format defines a standard container for the compressed data. In the frame format, the data is compressed into a sequence of blocks. The frame format defines a frame header, which contains information about the compressed data such as its size, and defines a standard end of frame marker.

The API provided by the frame format bindings follows that of the LZMA, zlib, gzip and bzip2 compression libraries which are provided with the Python standard library. As such, these LZ4 bindings should provide a drop-in alternative to the compression libraries shipped with Python. The package provides context managers and file handler support.

The bindings drop the GIL when calling in to the underlying LZ4 library, and is thread safe. An extensive test suite is included.

1.2 Install

The bindings to the LZ4 compression library provided by this package are in the form of a Python extension module written in C. These extension modules need to be compiled against the LZ4 library and the Python

1.2.1 Installing from pre-built wheels

The package is hosted on PyPI and pre-built wheels are available for Linux, OSX and Windows. Installation using a pre-built wheel can be achieved by:

```
$ pip install lz4
```

1.2.2 Installing from source

The LZ4 bindings require linking to the LZ4 library, and so if there is not a pre-compiled wheel available for your platform you will need to have a suitable C compiler available, as well as the Python development header files. On Debian/Ubuntu based systems the header files for Python are found in the distribution package `pythonX.Y-dev` e.g. `python3.7-dev`. On Fedora/Red Hat based systems, the Python header files are found in the distribution package `python-devel`.

The LZ4 library bindings provided by this package require the LZ4 library. If the system already has an LZ4 library and development header files present, and the library is a recent enough version the package will build against that. Otherwise, the package will use a bundled version of the library files to link against. The package currently requires LZ4 version 1.7.5 or later.

On a system for which there are no pre-built wheels available on PyPi, running this command will result in the extension modules being compiled from source:

```
$ pip install lz4
```

On systems for which pre-built wheels are available, the following command will force a local compilation of the extension modules from source:

```
$ pip install --no-binary --no-cache-dir lz4
```

The package can also be installed manually from a checkout of the source code git repository:

```
$ python setup.py install
```

Several packages need to be present on the system ahead of running this command. They can be installed using `pip`:

```
$ pip install -r requirements.txt
```

1.2.3 Test suite

The package includes an extensive test suite that can be run using:

```
$ python setup.py test
```

or, preferably, via `tox`:

```
$ tox
```

1.2.4 Documentation

The package also includes documentation in the `docs` directory. The documentation is built using Sphinx, and can be built using the included `Makefile`:

```
$ cd docs
$ make html
```

To see other documentation targets that are available use the command `make help`.

1.3 Quickstart

1.3.1 Simple usage

The recommended binding to use is the LZ4 frame format binding, since this provides interoperability with other implementations and language bindings.

The simplest way to use the frame bindings is via the *compress()* and *decompress()* functions:

```
>>> import os
>>> import lz4.frame
>>> input_data = 20 * 128 * os.urandom(1024)  # Read 20 * 128kb
>>> compressed = lz4.frame.compress(input_data)
>>> decompressed = lz4.frame.decompress(compressed)
>>> decompressed == input_data
True
```

The *compress()* function reads the input data and compresses it and returns a LZ4 frame. A frame consists of a header, and a sequence of blocks of compressed data, and a frame end marker (and optionally a checksum of the uncompressed data). The *decompress()* function takes a full LZ4 frame, decompresses it (and optionally verifies the uncompressed data against the stored checksum), and returns the uncompressed data.

1.3.2 Working with data in chunks

It's often inconvenient to hold the full data in memory, and so functions are also provided to compress and decompress data in chunks:

```
>>> import lz4.frame
>>> import os
>>> input_data = 20 * 128 * os.urandom(1024)
>>> c_context = lz4.frame.create_compression_context()
>>> compressed = lz4.frame.compress_begin(c_context)
>>> compressed += lz4.frame.compress_chunk(c_context, input_data[:10 * 128 * 1024])
>>> compressed += lz4.frame.compress_chunk(c_context, input_data[10 * 128 * 1024:])
>>> compressed += lz4.frame.compress_flush(c_context)
```

Here a compression context is first created which is used to maintain state across calls to the LZ4 library. This is an opaque PyCapsule object. *compress_begin()* starts a new frame and returns the frame header. *compress_chunk()* compresses input data and returns the compressed data. *compress_flush()* ends the frame and returns the frame end marker. The data returned from these functions is catenated to form the compressed frame.

compress_flush() also flushes any buffered data; by default, *compress_chunk()* may buffer data until a block is full. This buffering can be disabled by specifying `auto_flush=True` when calling *compress_begin()*. Alternatively, the LZ4 buffers can be flushed at any time without ending the frame by calling *compress_flush()* with `end_frame=False`.

Decompressing data can also be done in a chunked fashion:

```
>>> d_context = lz4.frame.create_decompression_context()
>>> d1, b, e = lz4.frame.decompress_chunk(d_context, compressed[:len(compressed)//2])
>>> d2, b, e = lz4.frame.decompress_chunk(d_context, compressed[len(compressed)//2:])
>>> d1 + d2 == input_data
True
```

Note that *decompress_chunk()* returns a tuple (decompressed_data, bytes_read, end_of_frame_indicator). decompressed_data is the decompressed data, bytes_read reports the number of bytes read from the compressed input. end_of_frame_indicator is True if the end-of-frame marker is encountered during the decompression, and False otherwise. If the end-of-frame marker is encountered in the input, no attempt is made to decompress the data after the marker.

Rather than managing compression and decompression context objects manually, it is more convenient to use the *LZ4FrameCompressor* and *LZ4FrameDecompressor* classes which provide context manager functionality:

```
>>> import lz4.frame
>>> import os
>>> input_data = 20 * 128 * os.urandom(1024)
>>> with lz4.frame.LZ4FrameCompressor() as compressor:
...     compressed = compressor.begin()
...     compressed += compressor.compress(input_data[:10 * 128 * 1024])
...     compressed += compressor.compress(input_data[10 * 128 * 1024:])
...     compressed += compressor.flush()
>>> with lz4.frame.LZ4FrameDecompressor() as decompressor:
...     decompressed = decompressor.decompress(compressed[:len(compressed)//2])
...     decompressed += decompressor.decompress(compressed[len(compressed)//2:])
>>> decompressed == input_data
True
```

1.3.3 Working with compressed files

The frame bindings provide capability for working with files containing LZ4 frame compressed data. This functionality is intended to be a drop in replacement for that offered in the Python standard library for bz2, gzip and LZMA compressed files. The *lz4.frame.open()* function is the most convenient way to work with compressed data files:

```
>>> import lz4.frame
>>> import os
>>> input_data = 20 * os.urandom(1024)
>>> with lz4.frame.open('testfile', mode='wb') as fp:
...     bytes_written = fp.write(input_data)
...     bytes_written == len(input_data)
True
>>> with lz4.frame.open('testfile', mode='r') as fp:
...     output_data = fp.read()
>>> output_data == input_data
True
```

The library also provides the class *lz4.frame.LZ4FrameFile* for working with compressed files.

1.3.4 Controlling the compression

Beyond the basic usage described above, there are a number of keyword arguments to tune and control the compression. A few of the key ones are listed below, please see the documentation for full details of options.

Controlling the compression level

The `compression_level` argument specifies the level of compression used with 0 (default) being the lowest compression (0-2 are the same value), and 16 the highest compression. Values below 0 will enable "fast acceleration", proportional to the value. Values above 16 will be treated as 16. The following module constants are provided as a convenience:

- *lz4.frame.COMPRESSIONLEVEL_MIN*: Minimum compression (0, default)
- *lz4.frame.COMPRESSIONLEVEL_MINHC*: Minimum high-compression mode (3)
- *lz4.frame.COMPRESSIONLEVEL_MAX*: Maximum compression (16)

Availability: *lz4.frame.compress()*, *lz4.frame.compress_begin()*, *lz4.frame.open()*, *lz4.frame.LZ4FrameCompressor*, *lz4.frame.LZ4FrameFile*.

Controlling the block size

The `block_size` argument specifies the maximum block size to use for the blocks in a frame. Options:

- *lz4.frame.BLOCKSIZE_DEFAULT* or 0: the lz4 library default
- *lz4.frame.BLOCKSIZE_MAX64KB* or 4: 64 kB
- *lz4.frame.BLOCKSIZE_MAX256KB* or 5: 256 kB
- *lz4.frame.BLOCKSIZE_MAX1MB* or 6: 1 MB
- *lz4.frame.BLOCKSIZE_MAX4MB* or 7: 4 MB

If unspecified, will default to *lz4.frame.BLOCKSIZE_DEFAULT* which is currently equal to *lz4.frame.BLOCKSIZE_MAX64KB*

Availability: *lz4.frame.compress()*, *lz4.frame.compress_begin()*, *lz4.frame.open()*, *lz4.frame.LZ4FrameCompressor*, *lz4.frame.LZ4FrameFile*.

Controlling block linking

The `block_linked` argument specifies whether to use block-linked compression. If `True`, the compression process will use data between sequential blocks to improve the compression ratio, particularly for small blocks. The default is `True`.

Availability: *lz4.frame.compress()*, *lz4.frame.compress_begin()*, *lz4.frame.open()*, *lz4.frame.LZ4FrameCompressor*, *lz4.frame.LZ4FrameFile*.

Data checksum validation

The `content_checksum` argument specifies whether to enable checksumming of the uncompressed content. If `True`, a checksum of the uncompressed data is stored at the end of the frame, and checked during decompression. Default is `False`.

The `block_checksum` argument specifies whether to enable checksumming of the uncompressed content of each individual block in the frame. If `True`, a checksum is stored at the end of each block in the frame, and checked during decompression. Default is `False`.

Availability: *lz4.frame.compress()*, *lz4.frame.compress_begin()*, *lz4.frame.open()*, *lz4.frame.LZ4FrameCompressor*, *lz4.frame.LZ4FrameFile*.

Data buffering

The LZ4 library can be set to buffer data internally until a block is filed in order to optimize compression. The `auto_flush` argument specifies whether the library should buffer input data or not.

When `auto_flush` is `False` the LZ4 library may buffer data internally. In this case, the compression functions may return no compressed data when called. This is the default.

When `auto_flush` is `True`, the compression functions will return compressed data immediately.

Availability: *lz4.frame.compress()*, *lz4.frame.compress_begin()*, *lz4.frame.open()*, *lz4.frame.LZ4FrameCompressor*, *lz4.frame.LZ4FrameFile*.

Storing the uncompressed source data size in the frame

The `store_size` and `source_size` arguments allow for storing the size of the uncompressed data in the frame header. Storing the source size in the frame header adds an extra 8 bytes to the size of the compressed frame, but allows the decompression functions to better size memory buffers during decompression.

If `store_size` is `True` the size of the uncompressed data will be stored in the frame header. Default is `True`.

Availability of `store_size`: *lz4.frame.compress()*

The `source_size` argument optionally specifies the uncompressed size of the source data to be compressed. If specified, the size will be stored in the frame header.

Availability of `source_size`: *lz4.frame.LZ4FrameCompressor.begin()*, *lz4.frame.compress_begin()*, *lz4.frame.open()*, *lz4.frame.LZ4FrameFile*.

1.3.5 Working with streamed compressed data

The stream bindings provide capability for working with stream compressed LZ4 data. This functionality is based on the usage of a ring-buffer (not implemented yet) or a double-buffer, with the length of each block preceding the compressed payload in the stream.

The stream compression reuses a context between each processed block for performance gain.

Most of the arguments used to initialize the LZ4 stream context are shared with the block API. Hereafter, those specific to the LZ4 stream API are detailed.

Controlling the buffer size

The `buffer_size` argument represents the base buffer size used internally for memory allocation:

- In the case of the double-buffer strategy, this is the size of each buffer of the double-buffer.

When compressing, this size is the maximal length of the input uncompressed chunks.

When decompressing, this size is the maximal length of the decompressed data.

Storing the compressed data size in the block

The `store_comp_size` argument allows tuning of the size (in bytes) of the compressed block, which is prepended to the actual LZ4 compressed payload. This size can be either on `1`, `2` or `4` bytes, or `0` for out-of-band block size record.

1.4 User Guide

1.4.1 lz4 package

Most of the functionality of this package is found in the *lz4.frame*, the *lz4.block* and the lz4.stream sub-packages.

Contents

lz4.**library_version_number**()
Returns the version number of the LZ4 library.

Parameters **None** –

Returns version number eg. 10705

Return type int

lz4.**library_version_string**()
Returns the version number of the LZ4 library as a string containing the semantic version.

Parameters **None** –

Returns version number eg. "1.7.5"

Return type str

1.4.2 lz4.frame sub-package

This sub-package is in beta testing. Ahead of version 1.0 there may be API changes, but these are expected to be minimal, if any.

This sub-package provides the capability to compress and decompress data using the LZ4 frame specification.

The frame specification is recommended for most applications. A key benefit of using the frame specification (compared to the block specification) is interoperability with other implementations.

Low level bindings for full content (de)compression

These functions are bindings to the LZ4 Frame API functions for compressing data into a single frame, and decompressing a full frame of data.

lz4.frame.**compress**()
compress(data, compression_level=0, block_size=0, content_checksum=0, block_linked=True, store_size=True, return_bytearray=False)

Compresses data returning the compressed data as a complete frame.

The returned data includes a header and endmark and so is suitable for writing to a file.

Parameters **data** (*str, bytes or buffer-compatible object*) – data to compress

Keyword Arguments

- **block_size** (*int*) – Sepcifies the maximum blocksize to use. Options:
  - *lz4.frame.BLOCKSIZE_DEFAULT*: the lz4 library default
  - *lz4.frame.BLOCKSIZE_MAX64KB*: 64 kB

- *lz4.frame.BLOCKSIZE_MAX256KB*: 256 kB
- *lz4.frame.BLOCKSIZE_MAX1MB*: 1 MB
- *lz4.frame.BLOCKSIZE_MAX4MB*: 4 MB

If unspecified, will default to *lz4.frame.BLOCKSIZE_DEFAULT* which is currently equal to *lz4.frame.BLOCKSIZE_MAX64KB*.

- **block_linked** (*bool*) – Specifies whether to use block-linked compression. If `True`, the compression ratio is improved, particularly for small block sizes. Default is `True`.
- **compression_level** (*int*) – Specifies the level of compression used. Values between 0-16 are valid, with 0 (default) being the lowest compression (0-2 are the same value), and 16 the highest. Values below 0 will enable "fast acceleration", proportional to the value. Values above 16 will be treated as 16. The following module constants are provided as a convenience:
  - *lz4.frame.COMPRESSIONLEVEL_MIN*: Minimum compression (0, the default)
  - *lz4.frame.COMPRESSIONLEVEL_MINHC*: Minimum high-compression mode (3)
  - *lz4.frame.COMPRESSIONLEVEL_MAX*: Maximum compression (16)
- **content_checksum** (*bool*) – Specifies whether to enable checksumming of the uncompressed content. If True, a checksum is stored at the end of the frame, and checked during decompression. Default is `False`.
- **block_checksum** (*bool*) – Specifies whether to enable checksumming of the uncompressed content of each block. If `True` a checksum of the uncompressed data in each block in the frame is stored at

  the end of each block. If present, these checksums will be used

  to validate the data during decompression. The default is `False` meaning block checksums are not calculated and stored. This functionality is only supported if the underlying LZ4 library has version >= 1.8.0. Attempting to set this value to `True` with a version of LZ4 < 1.8.0 will cause a `RuntimeError` to be raised.
- **return_bytearray** (*bool*) – If `True` a `bytearray` object will be returned. If `False`, a string of bytes is returned. The default is `False`.
- **store_size** (*bool*) – If `True` then the frame will include an 8-byte header field that is the uncompressed size of data included within the frame. Default is `True`.

Returns Compressed data

Return type bytes or bytearray

`lz4.frame.`**decompress**(*data*, *return_bytearray=False*, *return_bytes_read=False*)

Decompresses a frame of data and returns it as a string of bytes.

Parameters **data** (*str, bytes or buffer-compatible object*) – data to decompress. This should contain a complete LZ4 frame of compressed data.

Keyword Arguments

- **return_bytearray** (*bool*) – If `True` a bytearray object will be returned. If `False`, a string of bytes is returned. The default is `False`.
- **return_bytes_read** (*bool*) – If `True` then the number of bytes read from `data` will also be returned. Default is `False`

Returns

Uncompressed data and optionally the number of bytes read

If the `return_bytes_read` argument is `True` this function returns a tuple consisting of:

- bytes or bytearray: Uncompressed data
- int: Number of bytes consumed from `data`

Return type bytes/bytearray or tuple

Low level bindings for chunked content (de)compression

These functions are bindings to the LZ4 Frame API functions allowing piece-wise compression and decompression. Using them requires managing compression and decompression contexts manually. An alternative to using these is to use the context manager classes described in the section below.

Compression

`lz4.frame.`**`create_compression_context`**`()`
Creates a compression context object.

The compression object is required for compression operations.

Returns A compression context

Return type cCtx

`lz4.frame.`**`compress_begin`**`()`
compress_begin(context, source_size=0, compression_level=0, block_size=0, content_checksum=0, content_size=1, block_linked=0, frame_type=0, auto_flush=1)

Creates a frame header from a compression context.

Parameters **`context`** (*`cCtx`*) – A compression context.

Keyword Arguments

- **`block_size`** (*`int`*) – Sepcifies the maximum blocksize to use. Options:
  - *`lz4.frame.BLOCKSIZE_DEFAULT`*: the lz4 library default
  - *`lz4.frame.BLOCKSIZE_MAX64KB`*: 64 kB
  - *`lz4.frame.BLOCKSIZE_MAX256KB`*: 256 kB
  - *`lz4.frame.BLOCKSIZE_MAX1MB`*: 1 MB
  - *`lz4.frame.BLOCKSIZE_MAX4MB`*: 4 MB

  If unspecified, will default to *`lz4.frame.BLOCKSIZE_DEFAULT`* which is currently equal to *`lz4.frame.BLOCKSIZE_MAX64KB`*.
- **`block_linked`** (*`bool`*) – Specifies whether to use block-linked compression. If `True`, the compression ratio is improved, particularly for small block sizes. Default is `True`.
- **`compression_level`** (*`int`*) – Specifies the level of compression used. Values between 0-16 are valid, with 0 (default) being the lowest compression (0-2 are the same value), and 16 the highest. Values below 0 will enable "fast acceleration", proportional to the value. Values above 16 will be treated as 16. The following module constants are provided as a convenience:

- *lz4.frame.COMPRESSIONLEVEL_MIN*: Minimum compression (0, the default)
- *lz4.frame.COMPRESSIONLEVEL_MINHC*: Minimum high-compression mode (3)
- *lz4.frame.COMPRESSIONLEVEL_MAX*: Maximum compression (16)

- **content_checksum** (*bool*) – Specifies whether to enable checksumming of the uncompressed content. If True, a checksum is stored at the end of the frame, and checked during decompression. Default is `False`.
- **block_checksum** (*bool*) – Specifies whether to enable checksumming of the uncompressed content of each block. If `True` a checksum of the uncompressed data in each block in the frame is stored at

  the end of each block. If present, these checksums will be used

  to validate the data during decompression. The default is `False` meaning block checksums are not calculated and stored. This functionality is only supported if the underlying LZ4 library has version >= 1.8.0. Attempting to set this value to `True` with a version of LZ4 < 1.8.0 will cause a `RuntimeError` to be raised.
- **return_bytearray** (*bool*) – If `True` a `bytearray` object will be returned. If `False`, a string of bytes is returned. The default is `False`.
- **auto_flush** (*bool*) – Enable or disable autoFlush. When autoFlush is disabled the LZ4 library may buffer data internally until a block is full. Default is `False` (autoFlush disabled).
- **source_size** (*int*) – This optionally specifies the uncompressed size of the data to be compressed. If specified, the size will be stored in the frame header for use during decompression. Default is `True`
- **return_bytearray** – If `True` a bytearray object will be returned. If `False`, a string of bytes is returned. Default is `False`.

Returns Frame header.

Return type bytes or bytearray

lz4.frame.**compress_chunk**(*context*, *data*)

Compresses blocks of data and returns the compressed data.

The returned data should be concatenated with the data returned from *lz4.frame.compress_begin* and any subsequent calls to *lz4.frame.compress_chunk*.

Parameters

- **context** (*cCtx*) – compression context
- **data** (*str, bytes or buffer-compatible object*) – data to compress

Keyword Arguments **return_bytearray** (*bool*) – If `True` a bytearray object will be returned. If `False`, a string of bytes is returned. The default is False.

Returns Compressed data.

Return type bytes or bytearray

Notes

If auto flush is disabled (`auto_flush=False` when calling *lz4.frame.compress_begin*) this function may buffer and retain some or all of the compressed data for future calls to *lz4.frame.compress*.

lz4.frame.**compress_flush**(*context*, *end_frame=True*, *return_bytearray=False*)
Flushes any buffered data held in the compression context.

This flushes any data buffed in the compression context, returning it as compressed data. The returned data should be appended to the output of previous calls to `lz4.frame.compress_chunk`.

The `end_frame` argument specifies whether or not the frame should be ended. If this is `True` and end of frame marker will be appended to the returned data. In this case, if `content_checksum` was `True` when calling *lz4.frame.compress_begin*, then a checksum of the uncompressed data will also be included in the returned data.

If the `end_frame` argument is `True`, the compression context will be reset and can be re-used.

Parameters **context** (*cCtx*) – Compression context

Keyword Arguments

- **end_frame** (*bool*) – If `True` the frame will be ended. Default is `True`.
- **return_bytearray** (*bool*) – If `True` a `bytearray` object will be returned. If `False`, a `bytes` object is returned. The default is `False`.

Returns compressed data.

Return type bytes or bytearray

Notes

If `end_frame` is `False` but the underlying LZ4 library does not support flushing without ending the frame, a `RuntimeError` will be raised.

Decompression

lz4.frame.**create_decompression_context**()
Creates a decompression context object.

A decompression context is needed for decompression operations.

Returns A decompression context

Return type dCtx

lz4.frame.**reset_decompression_context**(*context*)
Resets a decompression context object.

This is useful for recovering from an error or for stopping an unfinished decompression and starting a new one with the same context

Parameters **context** (*dCtx*) – A decompression context

lz4.frame.**decompress_chunk**(*context*, *data*, *max_length=-1*)
Decompresses part of a frame of compressed data.

The returned uncompressed data should be concatenated with the data returned from previous calls to *lz4.frame.decompress_chunk*

Parameters

- **context** (*dCtx*) – decompression context
- **data** (*str, bytes or buffer-compatible object*) – part of a LZ4 frame of compressed data

Keyword Arguments

- **max_length** (*int*) – if non-negative this specifies the maximum number of bytes of uncompressed data to return. Default is `-1`.
- **return_bytearray** (*bool*) – If `True` a bytearray object will be returned.If `False`, a string of bytes is returned. The default is `False`.

Returns

uncompressed data, bytes read, end of frame indicator

This function returns a tuple consisting of:

- The uncompressed data as a `bytes` or `bytearray` object
- The number of bytes consumed from input `data` as an `int`
- The end of frame indicator as a `bool`.

Return type tuple

The end of frame indicator is `True` if the end of the compressed frame has been reached, or `False` otherwise

Retrieving frame information

The following function can be used to retrieve information about a compressed frame.

`lz4.frame.`**get_frame_info**(*frame*)

Given a frame of compressed data, returns information about the frame.

Parameters **frame** (*str, bytes or buffer-compatible object*) – LZ4 compressed frame

Returns

Dictionary with keys:

- `block_size` (int): the maximum size (in bytes) of each block
- `block_size_id` (int): identifier for maximum block size
- **content_checksum (bool): specifies whether the frame** contains a checksum of the uncompressed content
- `content_size` (int): uncompressed size in bytes of frame content
- `block_linked` (bool): specifies whether the frame contains blocks which are independently compressed (`False`) or linked linked (`True`)
- `block_checksum` (bool): specifies whether each block contains a checksum of its contents
- `skippable` (bool): whether the block is skippable (`True`) or not (`False`)

Return type dict

Helper context manager classes

These classes, which utilize the low level bindings to the Frame API are more convenient to use. They provide context management, and so it is not necessary to manually create and manage compression and decompression contexts.

class lz4.frame.**LZ4FrameCompressor**(*block_size=0*, *block_linked=True*, *compression_level=0*, *content_checksum=False*, *block_checksum=False*, *auto_flush=False*, *return_bytearray=False*)

Create a LZ4 frame compressor object.

This object can be used to compress data incrementally.

Parameters

- **block_size** (*int*) – Specifies the maximum blocksize to use. Options:
  - *lz4.frame.BLOCKSIZE_DEFAULT*: the lz4 library default
  - *lz4.frame.BLOCKSIZE_MAX64KB*: 64 kB
  - *lz4.frame.BLOCKSIZE_MAX256KB*: 256 kB
  - *lz4.frame.BLOCKSIZE_MAX1MB*: 1 MB
  - *lz4.frame.BLOCKSIZE_MAX4MB*: 4 MB

  If unspecified, will default to *lz4.frame.BLOCKSIZE_DEFAULT* which is equal to *lz4.frame.BLOCKSIZE_MAX64KB*.
- **block_linked** (*bool*) – Specifies whether to use block-linked compression. If `True`, the compression ratio is improved, especially for small block sizes. If `False` the blocks are compressed independently. The default is `True`.
- **compression_level** (*int*) – Specifies the level of compression used. Values between 0-16 are valid, with 0 (default) being the lowest compression (0-2 are the same value), and 16 the highest. Values above 16 will be treated as 16. Values between 4-9 are recommended. 0 is the default. The following module constants are provided as a convenience:
  - *lz4.frame.COMPRESSIONLEVEL_MIN*: Minimum compression (0)
  - *lz4.frame.COMPRESSIONLEVEL_MINHC*: Minimum high-compression (3)
  - *lz4.frame.COMPRESSIONLEVEL_MAX*: Maximum compression (16)
- **content_checksum** (*bool*) – Specifies whether to enable checksumming of the payload content. If `True`, a checksum of the uncompressed data is stored at the end of the compressed frame which is checked during decompression. The default is `False`.
- **block_checksum** (*bool*) – Specifies whether to enable checksumming of the content of each block. If `True` a checksum of the uncompressed data in each block in the frame is stored at the end of each block. If present, these checksums will be used to validate the data during decompression. The default is `False`, meaning block checksums are not calculated and stored. This functionality is only supported if the underlying LZ4 library has version >= 1.8.0. Attempting to set this value to `True` with a version of LZ4 < 1.8.0 will cause a `RuntimeError` to be raised.
- **auto_flush** (*bool*) – When `False`, the LZ4 library may buffer data until a block is full. When `True` no buffering occurs, and partially full blocks may be returned. The default is `False`.
- **return_bytearray** (*bool*) – When `False` a `bytes` object is returned from the calls to methods of this class. When `True` a `bytearray` object will be returned. The default is `False`.

begin(*source_size=0*)

Begin a compression frame.

The returned data contains frame header information. The data returned from subsequent calls to `compress()` should be concatenated with this header.

Keyword Arguments **source_size** (*int*) – Optionally specify the total size of the uncompressed data. If specified, will be stored in the compressed frame header as an 8-byte field for later use during decompression. Default is 0 (no size stored).

Returns frame header data

Return type bytes or bytearray

compress(*data*)

Compresses data and returns it.

This compresses `data` (a `bytes` object), returning a bytes or bytearray object containing compressed data the input.

If `auto_flush` has been set to `False`, some of `data` may be buffered internally, for use in later calls to *LZ4FrameCompressor.compress()* and *LZ4FrameCompressor.flush()*.

The returned data should be concatenated with the output of any previous calls to *compress()* and a single call to *compress_begin()*.

Parameters **data** (*str, bytes or buffer-compatible object*) – data to compress

Returns compressed data

Return type bytes or bytearray

flush()

Finish the compression process.

This returns a `bytes` or `bytearray` object containing any data stored in the compressor's internal buffers and a frame footer.

The LZ4FrameCompressor instance may be re-used after this method has been called to create a new frame of compressed data.

Returns compressed data and frame footer.

Return type bytes or bytearray

has_context()

Return whether the compression context exists.

Returns

True if the compression context exists, False otherwise.

Return type bool

reset()

Reset the *LZ4FrameCompressor* instance.

This allows the `LZ4FrameCompression` instance to be re-used after an error.

started()

Return whether the compression frame has been started.

Returns

True if the compression frame has been started, False otherwise.

Return type bool

class lz4.frame.**LZ4FrameDecompressor**(*return_bytearray=False*)

Create a LZ4 frame decompressor object.

This can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed frame at once, see *lz4.frame.decompress()*.

Parameters **return_bytearray** (*bool*) – When `False` a bytes object is returned from the calls to methods of this class. When `True` a bytearray object will be returned. The default is `False`.

eof

`True` if the end-of-stream marker has been reached. `False` otherwise.

Type bool

unused_data

Data found after the end of the compressed stream. Before the end of the frame is reached, this will be `b''`.

Type bytes

needs_input

`False` if the `decompress()` method can provide more decompressed data before requiring new uncompressed input. `True` otherwise.

Type bool

decompress(*data*, *max_length=-1*)

Decompresses part or all of an LZ4 frame of compressed data.

The returned data should be concatenated with the output of any previous calls to *decompress()*.

If `max_length` is non-negative, returns at most `max_length` bytes of decompressed data. If this limit is reached and further output can be produced, the *needs_input* attribute will be set to `False`. In this case, the next call to *decompress()* may provide data as `b''` to obtain more of the output. In all cases, any unconsumed data from previous calls will be prepended to the input data.

If all of the input `data` was decompressed and returned (either because this was less than `max_length` bytes, or because `max_length` was negative), the *needs_input* attribute will be set to `True`.

If an end of frame marker is encountered in the data during decompression, decompression will stop at the end of the frame, and any data after the end of frame is available from the *unused_data* attribute. In this case, the *LZ4FrameDecompressor* instance is reset and can be used for further decompression.

Parameters **data** (*str, bytes or buffer-compatible object*) – compressed data to decompress

Keyword Arguments **max_length** (*int*) – If this is non-negative, this method returns at most `max_length` bytes of decompressed data.

Returns Uncompressed data

Return type bytes

reset()

Reset the decompressor state.

This is useful after an error occurs, allowing re-use of the instance.

Reading and writing compressed files

These provide capability for reading and writing of files using LZ4 compressed frames. These are designed to be drop in replacements for the LZMA, BZ2 and Gzip equivalent functionalities in the Python standard library.

lz4.frame.**open**(*filename*, *mode='rb'*, *encoding=None*, *errors=None*, *newline=None*, *block_size=0*, *block_linked=True*, *compression_level=0*, *content_checksum=False*, *block_checksum=False*, *auto_flush=False*, *return_bytearray=False*, *source_size=0*)

Open an LZ4Frame-compressed file in binary or text mode.

`filename` can be either an actual file name (given as a str, bytes, or PathLike object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The `mode` argument can be `'r'`, `'rb'` (default), `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'`, or `'ab'` for binary mode, or `'rt'`, `'wt'`, `'xt'`, or `'at'` for text mode.

For binary mode, this function is equivalent to the *LZ4FrameFile* constructor: `LZ4FrameFile(filename, mode, ...)`.

For text mode, an *LZ4FrameFile* object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line ending(s).

Parameters **filename** (*str, bytes, os.PathLike*) – file name or file object to open

Keyword Arguments

- **mode** (*str*) – mode for opening the file
- **encoding** (*str*) – the name of the encoding that will be used for encoding/deconging the stream. It defaults to `locale.getpreferredencoding(False)`. See `io.TextIOWrapper` for further details.
- **errors** (*str*) – specifies how encoding and decoding errors are to be handled. See `io.TextIOWrapper` for further details.
- **newline** (*str*) – controls how line endings are handled. See `io.TextIOWrapper` for further details.
- **return_bytearray** (*bool*) – When `False` a bytes object is returned from the calls to methods of this class. When `True` a bytearray object will be returned. The default is `False`.
- **source_size** (*int*) – Optionally specify the total size of the uncompressed data. If specified, will be stored in the compressed frame header as an 8-byte field for later use during decompression. Default is 0 (no size stored). Only used for writing compressed files.
- **block_size** (*int*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **block_linked** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **compression_level** (*int*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **content_checksum** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **block_checksum** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **auto_flush** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.

class lz4.frame.**LZ4FrameFile**(*filename=None*, *mode='r'*, *block_size=0*, *block_linked=True*, *compression_level=0*, *content_checksum=False*, *block_checksum=False*, *auto_flush=False*, *return_bytearray=False*, *source_size=0*)

A file object providing transparent LZ4F (de)compression.

An LZ4FFile can act as a wrapper for an existing file object, or refer directly to a named file on disk.

Note that LZ4FFile provides a *binary* file interface - data read is returned as bytes, and data to be written must be given as bytes.

When opening a file for writing, the settings used by the compressor can be specified. The underlying compressor object is *lz4.frame.LZ4FrameCompressor*. See the docstrings for that class for details on compression options.

Parameters **filename** (*str, bytes, PathLike, file object*) – can be either an actual file name (given as a str, bytes, or PathLike object), in which case the named file is opened, or it can be an existing file object to read from or write to.

Keyword Arguments

- **mode** (*str*) – mode can be `'r'` for reading (default), `'w'` for (over)writing, `'x'` for creating exclusively, or `'a'` for appending. These can equivalently be given as `'rb'`, `'wb'`, `'xb'` and `'ab'` respectively.
- **return_bytearray** (*bool*) – When `False` a bytes object is returned from the calls to methods of this class. When `True` a `bytearray` object will be returned. The default is `False`.
- **source_size** (*int*) – Optionally specify the total size of the uncompressed data. If specified, will be stored in the compressed frame header as an 8-byte field for later use during decompression. Default is `0` (no size stored). Only used for writing compressed files.
- **block_size** (*int*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **block_linked** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **compression_level** (*int*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **content_checksum** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **block_checksum** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.
- **auto_flush** (*bool*) – Compressor setting. See *lz4.frame.LZ4FrameCompressor*.

close()

Flush and close the file.

May be called more than once without error. Once the file is closed, any other operation on it will raise a ValueError.

closed

Returns `True` if this file is closed.

Returns `True` if the file is closed, `False` otherwise.

Return type bool

fileno()

Return the file descriptor for the underlying file.

Returns file descriptor for file.

Return type file object

flush()
Flush the file, keeping it open.

May be called more than once without error. The file may continue to be used normally after flushing.

peek(*size=-1*)
Return buffered data without advancing the file position.

Always returns at least one byte of data, unless at EOF. The exact number of bytes returned is unspecified.

Returns uncompressed data

Return type bytes

read(*size=-1*)
Read up to `size` uncompressed bytes from the file.

If `size` is negative or omitted, read until `EOF` is reached. Returns `b''` if the file is already at `EOF`.

Parameters **size** (*int*) – If non-negative, specifies the maximum number of uncompressed bytes to return.

Returns uncompressed data

Return type bytes

read1(*size=-1*)
Read up to `size` uncompressed bytes.

This method tries to avoid making multiple reads from the underlying stream.

This method reads up to a buffer's worth of data if `size` is negative.

Returns `b''` if the file is at EOF.

Parameters **size** (*int*) – If non-negative, specifies the maximum number of uncompressed bytes to return.

Returns uncompressed data

Return type bytes

readable()
Return whether the file was opened for reading.

Returns

True if the file was opened for reading, False otherwise.

Return type bool

readline(*size=-1*)
Read a line of uncompressed bytes from the file.

The terminating newline (if present) is retained. If size is non-negative, no more than size bytes will be read (in which case the line may be incomplete). Returns b'' if already at EOF.

Parameters **size** (*int*) – If non-negative, specifies the maximum number of uncompressed bytes to return.

Returns uncompressed data

Return type bytes

seek(*offset*, *whence=0*)
Change the file position.

The new position is specified by `offset`, relative to the position indicated by `whence`. Possible values for `whence` are:

- `io.SEEK_SET` or 0: start of stream (default): offset must not be negative
- `io.SEEK_CUR` or 1: current stream position
- `io.SEEK_END` or 2: end of stream; offset must not be positive

Returns the new file position.

Note that seeking is emulated, so depending on the parameters, this operation may be extremely slow.

Parameters

- **offset** (*int*) – new position in the file
- **whence** (*int*) – position with which `offset` is measured. Allowed values are 0, 1, 2. The default is 0 (start of stream).

Returns new file position

Return type int

seekable()

Return whether the file supports seeking.

Returns `True` if the file supports seeking, `False` otherwise.

Return type bool

tell()

Return the current file position.

Parameters **None** –

Returns file position

Return type int

writable()

Return whether the file was opened for writing.

Returns

True if the file was opened for writing, False otherwise.

Return type bool

write(*data*)

Write a bytes object to the file.

Returns the number of uncompressed bytes written, which is always the length of data in bytes. Note that due to buffering, the file on disk may not reflect the data written until close() is called.

Parameters **data** (*bytes*) – uncompressed data to compress and write to the file

Returns the number of uncompressed bytes written to the file

Return type int

Module attributes

A number of module attributes are defined for convenience. These are detailed below.

Compression level

The following module attributes can be used when setting the `compression_level` argument.

`lz4.frame.`**`COMPRESSIONLEVEL_MIN`**
Specifier for the minimum compression level.

Specifying `compression_level=lz4.frame.COMPRESSIONLEVEL_MIN` will instruct the LZ4 library to use a compression level of 0

`lz4.frame.`**`COMPRESSIONLEVEL_MINHC`**
Specifier for the minimum compression level for high compression mode.

Specifying `compression_level=lz4.frame.COMPRESSIONLEVEL_MINHC` will instruct the LZ4 library to use a compression level of 3, the minimum for the high compression mode.

`lz4.frame.`**`COMPRESSIONLEVEL_MAX`**
Specifier for the maximum compression level.

Specifying `compression_level=lz4.frame.COMPRESSIONLEVEL_MAX` will instruct the LZ4 library to use a compression level of 16, the highest compression level available.

Block size

The following attributes can be used when setting the `block_size` argument.

`lz4.frame.`**`BLOCKSIZE_DEFAULT`**
Specifier for the default block size.

Specifying `block_size=lz4.frame.BLOCKSIZE_DEFAULT` will instruct the LZ4 library to use the default maximum blocksize. This is currently equivalent to *`lz4.frame.BLOCKSIZE_MAX64KB`*

`lz4.frame.`**`BLOCKSIZE_MAX64KB`**
Specifier for a maximum block size of 64 kB.

Specifying `block_size=lz4.frame.BLOCKSIZE_MAX64KB` will instruct the LZ4 library to create blocks containing a maximum of 64 kB of uncompressed data.

`lz4.frame.`**`BLOCKSIZE_MAX256KB`**
Specifier for a maximum block size of 256 kB.

Specifying `block_size=lz4.frame.BLOCKSIZE_MAX256KB` will instruct the LZ4 library to create blocks containing a maximum of 256 kB of uncompressed data.

`lz4.frame.`**`BLOCKSIZE_MAX1MB`**
Specifier for a maximum block size of 1 MB.

Specifying `block_size=lz4.frame.BLOCKSIZE_MAX1MB` will instruct the LZ4 library to create blocks containing a maximum of 1 MB of uncompressed data.

`lz4.frame.`**`BLOCKSIZE_MAX4MB`**
Specifier for a maximum block size of 4 MB.

Specifying `block_size=lz4.frame.BLOCKSIZE_MAX4MB` will instruct the LZ4 library to create blocks containing a maximum of 4 MB of uncompressed data.

1.4.3 lz4.block sub-package

This sub-package provides the capability to compress and decompress data using the block specification.

Because the LZ4 block format doesn't define a container format, the Python bindings will by default insert the original data size as an integer at the start of the compressed payload. However, it is possible to disable this functionality, and you may wish to do so for compatibility with other language bindings, such as the Java bindings.

Example usage

To use the lz4 block format bindings is straightforward:

```
>>> import lz4.block
>>> import os
>>> input_data = 20 * 128 * os.urandom(1024)  # Read 20 * 128kb
>>> compressed_data = lz4.block.compress(input_data)
>>> output_data = lz4.block.decompress(compressed_data)
>>> input_data == output_data
True
```

In this simple example, the size of the uncompressed data is stored in the compressed data, and this size is then utilized when uncompressing the data in order to correctly size the buffer. Instead, you may want to not store the size of the uncompressed data to ensure compatibility with the Java bindings. The example below demonstrates how to use the block format without storing the size of the uncompressed data.

```
>>> import lz4.block
>>> data = b'0' * 255
>>> compressed = lz4.block.compress(data, store_size=False)
>>> decompressed = lz4.block.decompress(compressed, uncompressed_size=255)
>>> decompressed == data
True
```

The `uncompressed_size` argument specifies an upper bound on the size of the uncompressed data size rather than an absolute value, such that the following example also works.

```
>>> import lz4.block
>>> data = b'0' * 255
>>> compressed = lz4.block.compress(data, store_size=False)
>>> decompressed = lz4.block.decompress(compressed, uncompressed_size=2048)
>>> decompressed == data
True
```

A common situation is not knowing the size of the uncompressed data at decompression time. The following example illustrates a strategy that can be used in this case.

```
>>> import lz4.block
>>> data = b'0' * 2048
>>> compressed = lz4.block.compress(data, store_size=False)
>>> usize = 255
>>> max_size = 4096
>>> while True:
...     try:
...         decompressed = lz4.block.decompress(compressed, uncompressed_size=usize)
...         break
...     except lz4.block.LZ4BlockError:
...         usize *= 2
...         if usize > max_size:
...             print('Error: data too large or corrupt')
...             break
>>> decompressed == data
True
```

In this example we are catching the `lz4.block.LZ4BlockError` exception. This exception is raisedd if the LZ4 library call fails, which can be caused by either the buffer used to store the uncompressed data (as set by `usize`) being too small, or the input compressed data being invalid - it is not possible to distinguish the two cases, and this is why we set an absolute upper bound (`max_size`) on the memory that can be allocated for the uncompressed data. If we did not take this precaution, the code, if ppassed invalid compressed data would continuously try to allocate a larger and larger buffer for decompression until the system ran out of memory.

Contents

`lz4.block.`**`compress`**(*source*, *mode='default'*, *acceleration=1*, *compression=0*, *return_bytearray=False*)

Compress source, returning the compressed data as a string. Raises an exception if any error occurs.

Parameters **`source`** (*`str, bytes or buffer-compatible object`*) – Data to compress

Keyword Arguments

- **`mode`** (*`str`*) – If `'default'` or unspecified use the default LZ4 compression mode. Set to `'fast'` to use the fast compression LZ4 mode at the expense of compression. Set to `'high_compression'` to use the LZ4 high-compression mode at the exepense of speed.
- **`acceleration`** (*`int`*) – When mode is set to `'fast'` this argument specifies the acceleration. The larger the acceleration, the faster the but the lower the compression. The default compression corresponds to a value of `1`.
- **`compression`** (*`int`*) – When mode is set to `high_compression` this argument specifies the compression. Valid values are between `1` and `12`. Values between `4-9` are recommended, and `9` is the default.
- **`store_size`** (*`bool`*) – If `True` (the default) then the size of the uncompressed data is stored at the start of the compressed block.
- **`return_bytearray`** (*`bool`*) – If `False` (the default) then the function will return a bytes object. If `True`, then the function will return a bytearray object.
- **`dict`** (*`str, bytes or buffer-compatible object`*) – If specified, perform compression using this initial dictionary.

Returns Compressed data.

Return type bytes or bytearray

`lz4.block.`**`decompress`**(*source*, *uncompressed_size=-1*, *return_bytearray=False*)

Decompress source, returning the uncompressed data as a string. Raises an exception if any error occurs.

Parameters **`source`** (*`str, bytes or buffer-compatible object`*) – Data to decompress.

Keyword Arguments

- **`uncompressed_size`** (*`int`*) – If not specified or negative, the uncompressed data size is read from the start of the source block. If specified, it is assumed that the full source data is compressed data. If this argument is specified, it is considered to be a maximum possible size for the buffer used to hold the uncompressed data, and so less data may be returned. If `uncompressed_size` is too small, `LZ4BlockError` will be raised. By catching `LZ4BlockError` it is possible to increase `uncompressed_size` and try again.
- **`return_bytearray`** (*`bool`*) – If `False` (the default) then the function will return a bytes object. If `True`, then the function will return a bytearray object.

- **dict** (*str, bytes or buffer-compatible object*) – If specified, perform decompression using this initial dictionary.

Returns Decompressed data.

Return type bytes or bytearray

Raises `LZ4BlockError` – raised if the call to the LZ4 library fails. This can be caused by `uncompressed_size` being too small, or invalid data.

1.4.4 lz4.stream sub-package

Warning: This module is unmaintained.

This sub-package is considered experimental. It was submitted by a community member who is not able to continue to maintain the module.

This module is not built as part of the distributed wheels. If you wish to build and use this module you will need to download and build from source with the environment variable PYLZ4_EXPERIMENTAL set to TRUE.

The module needs some re-write, and the tests need extensive work, for this to become production ready. If you are interested in working on this, please reach out to the package maintainers.

This sub-package provides the capability to compress and decompress data using the stream specification, especially the stream specification based on a double buffer.

Because the LZ4 stream format does not define a container format, the Python bindings will by default insert the compressed data size as an integer at the start of the compressed payload. However, it is possible to set the bit depth of this compressed data size.

So far, only the double-buffer based approach is implemented.

Example usage

To use the lz4 stream format bindings is straightforward:

```
>>> from lz4.stream import LZ4StreamCompressor, LZ4StreamDecompressor
>>> import os
>>> block_size_length = 2 # LZ4 compressed block size stored on 2 bytes
>>> page_size = 8192 # LZ4 context double buffer page size
>>> origin_stream = 10 * 1024 * os.urandom(1024) # 10MiB
>>> # LZ4 stream compression of origin_stream into compressed_stream:
>>> compressed_stream = bytearray()
>>> with LZ4StreamCompressor("double_buffer", page_size, store_comp_size=block_size_
↪length) as proc:
...     offset = 0
...     while offset < len(origin_stream):
...         chunk = origin_stream[offset:offset + page_size]
...         block = proc.compress(chunk)
...         compressed_stream.extend(block)
...         offset += page_size
>>> # LZ4 stream decompression of compressed_stream into decompressed_stream:
>>> decompressed_stream = bytearray()
>>> with LZ4StreamDecompressor("double_buffer", page_size, store_comp_size=block_size_
↪length) as proc:
...     offset = 0
```

(continues on next page)

(continued from previous page)

```
...     while offset < len(compressed_stream):
...         block = proc.get_block(compressed_stream[offset:])
...         chunk = proc.decompress(block)
...         decompressed_stream.extend(chunk)
...         offset += block_size_length + len(block)
>>> decompressed_stream == origin_stream
True
```

Out-of-band block size record example

```
>>> from lz4.stream import LZ4StreamCompressor, LZ4StreamDecompressor
>>> import os
>>> page_size = 8192 # LZ4 context double buffer page size
>>> out_of_band_block_sizes = [] # Store the block sizes
>>> origin_stream = 10 * 1024 * os.urandom(1024) # 10MiB
>>> # LZ4 stream compression of origin_stream into compressed_stream:
>>> compressed_stream = bytearray()
>>> with LZ4StreamCompressor("double_buffer", page_size, store_comp_size=0) as proc:
...     offset = 0
...     while offset < len(origin_stream):
...         chunk = origin_stream[offset:offset + page_size]
...         block = proc.compress(chunk)
...         out_of_band_block_sizes.append(len(block))
...         compressed_stream.extend(block)
...         offset += page_size
>>> # LZ4 stream decompression of compressed_stream into decompressed_stream:
>>> decompressed_stream = bytearray()
>>> with LZ4StreamDecompressor("double_buffer", page_size, store_comp_size=0) as proc:
...     offset = 0
...     for block_len in out_of_band_block_sizes:
...         # Sanity check:
...         if offset >= len(compressed_stream):
...             raise LZ4StreamError("Truncated stream")
...         block = compressed_stream[offset:offset + block_len]
...         chunk = proc.decompress(block)
...         decompressed_stream.extend(chunk)
...         offset += block_len
>>> decompressed_stream == origin_stream
True
```

Contents

1.5 Contributors

- Jonathan Underwood combined the block and frame modules into a coherent single project with many fixes, clean-ups and documentation
- Jonathan Underwood added frame bindings based on the lz4ex by Jerry Ryle and the lz4tools project by Christopher Jackson
- Jonathan Underwood updated the block format support to use the tunable accelerated and high compression functions
- Mathew Rocklin added support for dropping the GIL to the block module, and Travis testing support

- Antoine Martin added initial support for fast compression support to the block library
- Steve Morin wrote the original lz4 block bindings

1.6 Licensing

Code specific to this project is covered by the BSD 3-Clause License

CHAPTER 2

Indices

- genindex
- modindex
- search

Python Module Index

l

Index

S

T

U

W